MATH 446, DATA SCIENCE WITH PYTHON, FALL 2024

STEVEN HEILMAN

Contents

Date: November 24, 2024

1. Python, IPython, Packages

Python is a freely available software. You should download and install this software on your personal computer. Specifically, download Anaconda (a popular Python distribution platform) from here: https://www.anaconda.com/download. Instructions for downloading and installing this software can be found: here. It might be helpful to bring a laptop to class with Python installed. Students who do not own a laptop may consider the USC Laptop Loaner Program: https://itservices.usc.edu/spaces/laptoploaner/.

We will most commonly be using the Jupyter notebook, within Anaconda.

Once you open Jupyter Notebook, you can run a block of code by typing some commands, holding shift and pressing enter (or pressing the "play" button).

Jupyter Notebok runs IPython, an enhanced Python interpreter.

We will be using some widely used packages in this course. Packages contain pre-written programs. Make sure the following packages are installed in Anaconda:

- Numpy (Numerical Python): best for homogeneous (numerical) data types
- Pandas: data structures and data manipulation, best for heterogeneous data types
- SciPy: optimization, linear algebra, integration, etc.
- Matplotlib: data visualization
- Scikit-learn (sklearn): machine learning

Time permitting, we might be able to cover some tools from the following packages:

- keras: deep learning library that interfaces with the following two packages:
- TensorFlow: Google's machine learning library
- PyTorch: Meta's machine learning library

Although we will be learning about these packages, we will also try to understand how they work, instead of relying on the packages to do all of the work.

It is considered good practice to only import packages that you will use. At the start of a script, use the command `import numpy` to import the Numpy package.

Coding Convention. We will be using the Numpy and pandas packages so frequently that we will **always assume** every block of code we write is preceded by the following commonly used commands:

```
import numpy as np
import pandas as pd
```

These commands let us use packages succinctly, e.g. `np.pi` is shorter than `numpy.pi`.

If you want to brush up on the basics of coding in Python, I recommend this website.

Coding Style. Since Python is an open source language, the Python coding community has adopted a coding style that one should try to follow. For more details on these style specifics, see e.g. here and here. Whenever possible, try to follow this style. For example, every equals sign should have one space on its left and one space on its right. Functions and variable names should be descriptive, with lower case words separated by underscores (`basketball_data` could be a matrix of basketball data, rather than `bdata`), etc.

You might prefer adding comments with # symbols, though style dictates that comments should appear above or below lines of code, rather than to the right of a code line. We can also add comments by changing a coding cell in Jupyter to Markdown (using a dropdown menu to the right of the "fast forward" symbol). In a Markdown cell, the commands #, ## and ### can produce ever larger text headings.

1.1. Introduction.

As an introduction to Python let's begin with some basic syntax. Arithmetic operations such as `1 + 4` or `5 - 2.5` produce their expected outputs. Multiplication uses the asterisk symbol, so that `3 * 6` evaluates to 18. Strings can be surrounded by single or double quotes. Strings can be multiplied, e.g. `'s'*4` outputs `'ssss'`. Also `6 / 3` evaluates to 2. Exponents use the `**` symbol, so `2**3` evaluates to 8. The irrational number π is built into the numpy package, so that typing `np.pi` and pressing shift-enter results in

$$3.141592653589793.$$

Variables are defined using the equals sign[1]. For example, `x = 2` assigns the value 2 to the variable x. With this assignment, `3 * x` produces the output 6. Vectors can also be assigned to variable names: `x = np.array([2,   3])` assigns the array $(2, 3)$ to the variable name x. Within Python, this array is a list of numbers that is not explicitly identified as a row vector or as a column vector, as we might do in a linear algebra class. (The array can be viewed with the command `print(x)` .) If we wanted to represent x as a row vector `row_x`, we could use the command `row_x = x[np.newaxis, :]`. Similarly, `col_x = x[:, np.newaxis]` assigns the column vector $\binom{2}{3}$ to the variable `col_x`. To see the difference between these objects, observe that the commands `x.shape`, `x_row.shape` and `x_col.shape` have outputs $(2,)$, $(1, 2)$ and $(2, 1)$, respectively.

The attributes of an object in Python can be found with the `?` command. For example, `x?` returns the attributes of x. (Also basically everything in Python is an object: numbers, strings, data structures, functions, classes, modules, etc. are all objects.)

The zeroth entry[2] of a vector x can be accessed with the command `x[0]`. A vector or matrix can be transposed with the `np.transpose` command. For example, `np.transpose(x_row)` returns a column vector identical to `x_col`. Alternatively, `x_row.T` is also the transpose of `x_row`.

A 2×3 matrix can be created with the command

```
x = np.array([ [3, 1, 2], [5, 3, 6] ])
```

Syntactically, this 2×3 matrix is entered into Python as an array of two length three arrays, hence the nested brackets. The command `x.shape` returns $(2, 3)$, denoting that `x` is a 2×3 matrix. However, the "array of arrays" structure is also reflected in `x[0]` evaluating to `[3,1,2]`, the first row of the matrix. The $(1, 2)$ entry of x (i.e. 6) can be accessed as either `x[1][2]` or `x[1, 2]`. Sub-arrays can be accessed by e.g. `x[:, :2]`, which outputs the matrix $\begin{pmatrix} 3 & 1 \\ 5 & 3 \end{pmatrix}$, as does `x[:, 0:2]`

WARNING. Slicing, like the `range` command, uses half open intervals. For example, if `z = np.array([8, 7, 6, 5])`, then `z[1:3]` will output all entries between and including indices 1 and 2 (but not 3), i.e. the output is $(7, 6)$. Similarly, `x[1:2, 1:2]` outputs 3, which is identical to `x[1, 1]`.

[1]Python allows you to assign a number value to a variable x, and then assign a vector value to x, and then assign a number value to x. Other programming languages do not allow variables to change types.

[2]Other programming languages might denote the initial entry of a vector as the 1 entry.

Matrix multiplication can be done with the `@` symbol. If `y = np.array([[1, 2], [1, 3]])`, then $y@y$ returns

$$\begin{pmatrix}1 & 2\\ 1 & 3\end{pmatrix}\begin{pmatrix}1 & 2\\ 1 & 3\end{pmatrix} = \begin{pmatrix}1\cdot 1+2\cdot 1 & 1\cdot 2+2\cdot 3\\ 1\cdot 1+3\cdot 1 & 1\cdot 2+3\cdot 3\end{pmatrix} = \begin{pmatrix}3 & 8\\ 4 & 11\end{pmatrix}.$$

Numpy syntax streamlines vector and matrix operations. For example, `2*np.array([3,4])` evaluates to `[6,8]`. Component-wise operations borrow syntax from the arithmetic of real numbers:

- `np.array([2, 3]) + np.array([4, 5])` evaluates to `[6, 8]`.
- `np.array([6, 8]) / np.array([2, 4])` evaluates to `[3, 2]`.
- `np.array([6, 8]) * np.array([2, 4])` evaluates to `[12, 32]`.

The last command should not be confused with the dot product of two vectors, such as `np.dot(np.array([6, 8]), np.array([2, 4]))`, which evaluates to $6\cdot 2+8\cdot 4 = 12+32 = 44$.

Matrix powers of `x=np.array([[1, 2], [1, 3]])` can be computed as follows: `np.linalg.matrix_power(x,3)` has output

$$\begin{pmatrix}1 & 2\\ 1 & 3\end{pmatrix}^3 = \begin{pmatrix}11 & 30\\ 15 & 41\end{pmatrix}.$$

The `%` command takes a modulus, so `5 % 2` outputs 1, and `//` is floor division, so `7 // 2` outputs 3.

Sub-arrays can be accessed in the following way.

```
x = np.array([4, 6, 8, 9, 3])
y = x[2:4]
print(y)
```

The output of this program is the array $(8, 9)$. If we then use the command `y[0]=2`, then `print(y)` returns the array $(2, 9)$. Moreover, `print(x)` returns the array $(4, 6, 2, 9, 3)$. That is, changing the value of y also changes the value of x. In Python, the sub-array y is considered a sub-object of y, and changes to y are inherited by x. Moreover, y is never allocated its own memory as an array distinct of x.

1.2. **Data Structures.** A **tuple** is a fixed-length, immutable ordered sequence of Python objects. For example `tup = (4, 5, 6)` creates a tuple with three elements, and so does `tup = 4, 5, 6` or `tup = tuple([4, 5, 6])`. A string can also be converted to a tuple: `tuple("foo")` outputs the tuple `("f", "o", "o")`. Tuple elements can be accessed with squared brackets, so `tup[0]` outputs 4 and `tup[1]` outputs 5. Objects inside tuples can be modified, e.g.

```
tup = ("foo", [4, 5], 5)
tup[1].append(6)
```

outputs `("foo", [4, 5, 6], 5)`. Tuples can be concatenated with the `+` sign. As with lists or strings, multiplying a tuple by a positive integer n will concatenate it with itself n times: `(3, 4, 5)*3` outputs `(3, 4, 5, 3, 4, 5, 3, 4, 5)`. Tuples can be unpacked with assignments. For example, `a, b, c = tup` outputs `a = 4`, `b = 5` and `c = 6`. We can e.g. swap variables this way with the command `b, a = a, b`.

Immutable data types include: `int`, `float`, `complex`, `bool`, `tuple`, `str`, `None`. With the commands `x = 5` followed by `x = 6`, we can change the value of x, but since integers are immutable, the original x is removed from memory and replaced by the new x value. Similarly, we can assign a tuple to x, but the tuple itself cannot be changed. Mutable (not immutable) data types include: `list`, `set` and `dict`. The following function will tell you whether or not something is immutable in Python (except for tuples of integers).

```
def is_immutable(x):
    if type(x) in (list, dict, set):
        print(f"{x} mutable")
    else:
        print(f"{x} is immutable")
```

(The indentations here are part of proper syntax. The standard indentation is four spaces. IPython will automatically convert a Tab to four spaces, in case you are used to using tabs instead of spaces.)

Immutable data types must be used for elements of sets or keys in dictionaries. For example, Python disallows set elements to be lists: `{[1, 2], 3}` produces an exception, though `{(1, 2), 3}` is allowed. Immutable data types can be considered more "reliable," since the data cannot change. In contrast, mutable data types are memory efficient, since e.g. updating a few entries in a list does not require creating an entirely new list in memory.

In the following example, we iterate over a list of tuples and print out the results. The `f` at the beginning of the string indicates a formatted string, which allows the variables a, b, c to be substituted into the string at the positions specified by the curly brackets $\{\cdots\}$.

```
seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
for a, b, c in seq:
    print(f'x={a}, y={b}, z={c}')
```

outputs

```
x=1, y=2, z=3
x=4, y=5, z=6
x=7, y=8, z=9
```

Besides a formatted string, a raw string with `r` prefix will treat `\` as a literal rather than an escape character. For example, `print(r"foo\nfop")` and `print("foo\\nfop")` both print `foo\nfop`, whereas `print("foo\nfop")` will print

```
foo
fop
```

since the `\n` commands denotes a line break.

In the following example, the `rest` variable is assigned a list of the values after a and b.

```
values = 1, 2, 3, 4, 5
a, b, *rest = values
```

So, `rest = [3, 4, 5]`, although `values` is a tuple.

If we only care about the variables a, b, we could also use the command

```
a, b, *_ = values
```

to make no assignment of values after 1 and 2. The underscore can be used in other instances (e.g. for loops) when you do not want to assign a variable to a value.

A **list** is a variable-length, mutable sequence of Python objects, e.g. `lis = [4, 5, 6]` creates a list with three elements, as does `lis = list((4, 5, 6))`. Tuple elements can be accessed with squared brackets, so `lis[0]` outputs 4 and `lis[1]` outputs 5. The command `ran = range(5)` or `ran = range(0, 5)` is similar but distinct from a list of the sequence $0, 1, 2, 3, 4$. Unlike tuples, lists can be modified, e.g. `lis.append(7)` results in `lis` being `[4, 5, 6, 7]`. To insert at a specific entry of the list, we can use `lis.insert(2, 5.4)` to get `lis` that is `[4, 5, 5.4, 6, 7]`. However, the `insert` command is more computationally expensive than appending. The `pop` function removes the stated entry of a list: `lis.pop(0)` returns 4, and then `lis` takes the form `[5, 5.4, 6, 7]`. The command `lis.remove(5)` makes `lis` have the form `[5.4, 6, 7]`. This command removes the stated entry of the list (with the smallest index in case of ties). We can check for membership of an element in a list; e.g. `6 in lis` returns `True` and `6 not in lis` returns `False`.

Lists can be concatenated with the `+` operation. Multiple elements can be added to a list with the `extend` function. Lists can be sorted with the `sort` function. List elements can be sliced with `:`. For example

```
lis3 = [6, 7, 4, 1, 9, 6, 7, 0, 10]
print(lis3[1:4])
print(lis3[4:])
print(lis3[:2])
print(lis3[-1:3])
```

outputs

```
[7, 4, 1]
[9, 6, 7, 0, 10]
[6, 7]
[1, 9, 6, 7]
```

A **dictionary** is a bijection between two sequences of objects, sometimes called a hash map or an associative array. A dictionary is instantiated with curly brackets, e.g.

```
diction = {"a" : 5, "c" : 7, "e" : "nine"}
```

is a dictionary with two sequences of length three, where `dict["a"]` returns 5, `dict["c"]` returns 7 and `dict["e"]` returns “nine”. The sequence a, b, e can be called **keys** and the sequence $5, 7$, nine can be called **values**, so a dictionary is a collection of key-value ordered pairs. (The keys of the dictionary must be immutable, e.g. a key value cannot be a list.) We can create new elements of a dictionary with an assignment `diction["f"] = 10` gives

```
{'a': 5, 'c': 7, 'e': 'nine', 'f': 10}
```

The `pop` function can remove elements of the dictionary, e.g. `diction.pop("c")` returns 7 and then `diction` takes the form `{'a': 5, 'e': 'nine', 'f': 10}`. Similarly, `del diction["c"]` deletes that same entry from the dictionary. To output the keys and values separately as lists, we can use `list(diction.keys())` and `list(diction.values())`. Two dictionaries can be merged with the `update` function, so

```
diction.update({"c": 8, "e": "n", "g": 12})
```

outputs

```
{'a': 5, 'e': 'n', 'f': 10, 'c': 8, 'g': 12}
```

Note that the updated values take the place of old values. A dictionary can be created with the `zip` function.

```
dict(zip(range(5), reversed(range(5))))
```

outputs

```
{0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

A **set** is an unordered collection of unique immutable elements. A set can be created with the `set` function or with curly braces, e.g. `set([1, 3, 2, 4, 2, 1])` outputs `{1, 2, 3, 4}`, as does `{1, 3, 2, 4, 2, 1}`. We can take the union of two sets using the `union` function or the `|` symbol.

```
{1, 2, 3}.union({2, 3, 4})
```

outputs

```
{1, 2, 3, 4}
```

as does `{1, 2, 3}|{2, 3, 4}` We can take the intersection of two sets using the `intersection` function or the `&` symbol.

```
{1, 2, 3}.intersection({2, 3, 4})
```

outputs

```
{2, 3}
```

as does `{1, 2, 3}&{2, 3, 4}`. Set containment can be checked with the `issubset` or `issuperset` functions. Set equality can be checked with the `==` symbol, so

```
{1, 2 ,3} == {2, 3, 1}
```

returns `True`.

1.2.1. *Useful Built-in Sequence Functions.* If `seq` is a sequence of objects, then `enumerate(seq)` outputs an iterable (i.e. used in a for loop) set of tuples of the form $(0, s_0), (1, s_1), \ldots, (n, s_n)$ where $s_0, \ldots, s_n$ are the elements of `seq`.

```
for index, value in enumerate({4, 5, 6}):
    print(f"index = {index}, value = {value}")
```

outputs

```
index = 0, value = 4
index = 1, value = 5
index = 2, value = 6
```

Also, `list(enumerate({4, 5, 6}))` outputs `[(0, 4), (1, 5), (2, 6)]`.

The `sorted` function returns a sorted list when given an input sequence of elements.

The `zip` function outputs an iterable sequence of length k tuples when given an input of k sequences of elements. The output's length is the shortest input sequence length. For example,

```
list(zip([1, 3, 4], ("a", "b", "c", "t")))
```

outputs

```
[(1, 'a'), (3, 'b'), (4, 'c')]
```

The `reversed` function outputs an iterable sequence in reversed order.

The functions `enumerate`, `sorted`, `reversed` are all generator objects, i.e. they output one element at a time to save memory. You can create your own generator object by replacing a `return` command in a function with a `yield` command. (Technically a generator object should be implemented in Python with a `yield` command, so you could say `enumerate` is not a generator object since it is implemented in C, but we will not make such a distinction in these notes.)

1.2.2. *Comprehensions.* A comprehension is basically a one-line for loop. Comprehensions are a convenient way to apply functions over sequences. The second line of the following code is an example of a list comprehension.

```
strings = ["a", "as", "bat", "car", "dove", "python"]
[x.upper() for x in strings if len(x) > 2]
```

outputs

```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

The list comprehension is equivalent to the following for loop:

```
strings = ["a", "as", "bat", "car", "dove", "python"]
output = []
for string in strings:
    if len(string)>2:
        output.append(string.upper())
print(output)
```

Dictionary and set comprehensions can be created analogously. However, if we take a list comprehension and replace the outer square brackets with parentheses, we will get a generator expression (i.e. we will not get a tuple comprehension).

List comprehensions can also be nested. Before nesting, the following block of code outputs all names in a list with at least two "a" characters.

```
all_data = [["Ezequiel", "Denver", "John", "Janiyah", "Vihaan"],
            ["Maria", "Juan", "Javier", "Natalia", "Ainhoa"]]
names_of_interest = []
for name_list in all_data:
    enough_a = [name for name in name_list if name.count("a") >= 2]
    names_of_interest.extend(enough_a)
names_of_interest
```

outputs

```
['Janiyah', 'Vihaan', 'Maria', 'Natalia']
```

The for loop can be combined with the comprehension to get a nested list comprehension:

```
[name for name_list in all_data for name in name_list if name.count("a") >= 2]
```

The inner line `for name in name_list if name.count("a") >= 2` also appears in the original code block we wrote above, though in the nested comprehension we put the outer for loop as another comprehension on the left side.

The nested comprehension should be contrasted with the slightly different comprehension put inside another one.

```
[[x for x in name_list if x.count("a") >=2 ] for name_list in all_data]
```

whose output is

```
[['Janiyah', 'Vihaan'], ['Maria', 'Natalia']]
```

1.3. **Functions.** The polynomial

$$f(x) = x^2 - x - 1, \qquad \forall\, x \in \mathbb{R}$$

is quadratic with two real zeros. (From the quadratic formula, f as zeros at $\frac{1\pm\sqrt{1+4}}{2} = \frac{1\pm\sqrt{5}}{2}$.) Here is syntax for a function definition:

```
def myfun(x):
    return x**2 - x - 1
```

After making this definition, we can call this function. That is, `myfun(0)` returns -1 and `myfun(2)` returns 1. We can also rename this function with e.g. `f = myfun`, so that `f(0)` returns -1 and `f(2)` returns 1. Since math expressions work for numpy array inputs, we can also input numpy arrays to f, so that `f(np.array([0, 2]))` returns `[-1, 1]`. We can plot this function using the `matplotlib` package:

```
import matplotlib.pyplot as plt
x = np.arange(-1, 1, .02)
plt.plot(x, myfun(x))
plt.show()
```

Grid lines can be added to the plot with the `plt.grid(True)` command. The axes can be defined by e.g. `plt.axis([-1, 1, -2, 2])`. Axes can be labelled with commands `plt.xlabel('horizontal axis')` and `plt.ylabel('vertical axis')` .

Here the `arange` function outputted the points $-1, -1+.02, -1+.04, -1+.06, \ldots, 1-.02$, i.e. evenly spaced consecutive points with spacing .02 on the interval $[-1, 1)$.

Functions can have multiple arguments and optional arguments with default values, e.g.

```
def myfun2(x, y, z = 4):
    return x**2 - y - z
print(
    myfun2(1, 1),
    myfun2(1, 1, 3),
    myfun2(1, 1, z = 3)
)
```

outputs -4, -3, -3. Optional (keyword) arguments must appear to the right of non-optional (positional) arguments. Multiple outputs can be used with a single `return` statement.

Python allows you to make short single-line functions that are inputs into other functions using the `lambda` keyword. These functions are called anonymous functions. For example, we could rewrite `myfun` as an anonymous function with

```
myfun_anonymous = lambda x: x**2 - x - 1
```

Below is an example demonstrating the use of an anonymous function.

```
def apply_to_list(some_list, func):
    return [func(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

which outputs

```
[8, 0, 2, 10, 12]
```

Functions in the numpy package include: `sin, cos, tan, log, exp`. For-loops use the following syntax:

```
for i in range(4):
    print(i)
```

This will print the integers from 0 to 3 in increasing order. Here `range(4)` is a sequence of integers from 0 to 3. (The output of the `range` function is not an array. Its data type is “range,” which is similar to a generator.)

Python for-loops can also iterate over items of any sequence:

```
words = ['cat', 'window', 'dogs']
for w in words:
    print(w, len(w))
```

This code has output

```
cat 3
window 6
dogs 4
```

While-loops use the following syntax:

```
i = 1
while i<10:
    print(i)
    i = i + 1
```

Single conditionals use the following syntax:

```
if x<10:
    print(x)
else:
    print(x + 1)
```

Multiple conditionals use the following syntax:

```
if x<10:
    print(x)
elif 10<= x <12:
    print(x + 1)
elif 12<= x <13:
    print(x + 2)
else:
    print(x + 3)
```

There are a few ways to measure the time of parts of Python code. For a small bit of code, we can use the `%timeit` special function with syntax `%timeit 5*2` to get an output of

```
5.62 ns ± 0.139 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)
```

That is, multiplying 5 times 2 takes approximately 5.62 nanoseconds, i.e. 5.62×10^{-9} seconds.

Alternatively, we can use the `time` package to set various time “checkpoints”, and then measure their differences.

```
import time
first_time = time.time()
time.sleep(1)
second_time = time.time()
time.sleep(2)
third_time = time.time()
print(third_time - first_time)
print(second_time - first_time)
```

The output from this code is

```
3.0013132095336914
1.0004501342773438
```

When running a for loop for a long time, it can sometimes be helpful to check its progress. The `tqdm` package adds a progress bar to such a for loop with the following syntax.

```
from tqdm import tqdm
import time
for i in tqdm(range(100)):
    time.sleep(.1)
```

Exercise 1.1. In Python, do the following:

- Perform the following operation, and report the result:

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + 4 \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}.$$

- Plot the function $f(x) = x^3 + e^x$ for x values in the interval $[0, 3]$.
- Describe the output of the following program.

```
x = 1
while x != 0:
    x = x / 2
    print(x)
```

Exercise 1.2. In Python, the logical value `True` represents true, and the logical value `False` represents a false statement. For example, `3<5` evaluates to a `True`, and `5<3` evaluates to `False`.

Python's logical operations include: `and` for logical and, `or` for logical or, `not` for logical negation. Python's relational operations include: `<` for less than, `<=` for less than or equal to, `==` for equality, `!=` for not equality. (The command `&` is logical and that also works for arrays. The command `|` is logical or that also works for arrays.)

- Compute the following expression by hand, and in Python:

```
( (2<3) and (4<2) ) or not(4<8).
```

- Describe the output of the following program.

```
x=1
while (x<5) and not(x< -5):
    x = x + np.random.random()
    print(x)
```

- Logical operations can also apply to vectors, using Numpy functions. Compute the output of the following program by hand, and in Python:

```
x = np.array([False, True, False])
y = np.array([True, True, False])
z = np.array([False, False, True])
a = (x & y) | z
print(a)
```

(Note: Numpy's logical arrays can also be summed, where True acts as 1 and False acts as 0, so the sum of [True, True, False] would be 2.)

1.3.1. *Opening and Closing Files.* In the discussion below, I first save the following .txt file as `myfile.txt` .

```
Here is a text file,
to be used for testing
stuff in Python.
```

To open this file, we can use the following command.

```
fil = open(r"C: ...\myfile.txt, encoding = "utf-8")
```

For convenience I use a raw string so I can write single \ characters. Also, the optional `encoding = "utf-8"` uses UTF-8 encoding (UTF-8 is a standard 8-bit (1 byte) character encoding, which is typically but not always the default encoding choice.) We can print this file e.g. using:

```
for line in fil:
    print(line)
```

To save memory, it is recommended to close any opened file after the file is used, using

```
fil.close()
```

Alternatively, you can automatically close a file by nesting it within a `with` statement.

```
with open(r"C: ...\myfile.txt", encoding = "utf-8") as fil:
    lines = [x.rstrip() for x in fil]
```

which outputs

```
['Here is a text file,', 'to be used for testing', 'stuff in Python.']
```

A portion of a file can be read using `fil.read(10)` will output ten characters from `fil`. `fil.tell()` will output the current file's reading position. `fil.seek(3)` will change the file reading position to the third byte.

2. Floating Point Number System

Definition 2.1. The most common number system used on computers is the **double precision floating point** system. This number system includes any number of the form

$$\pm(1.a_1a_2\cdots a_{52})\cdot 2^{b_{11}\cdots b_1-1023}=\pm\Big(1+\sum_{i=1}^{52}2^{-i}a_i\Big)\cdot 2^{\sum_{j=0}^{10}2^jb_{j+1}-1023},$$

where $a_1,\ldots,a_{52},b_1,\ldots,b_{11}\in\{0,1\}$ are binary digits, **and** $b_1,\ldots,b_{11}$ are not all 0 and they are not all 1. Numbers of this form are called **normal numbers**. The 52-bit binary number $.a_1\cdots a_{52}$ is called the **mantissa**, and the 11-bit exponent $b_{11}\cdots b_1-1023$ is called the **exponent** of the floating point number. One bit is need to store the sign (+ or −) for a total of $52+11+1=64$ bits.

In Python, the binary representation of $(-1)^c\cdot(1.a_1a_2\cdots a_{52})\cdot 2^{b_{11}\cdots b_1-1023}$ with $c\in\{0,1\}$ is ordered as

$$cb_{11}b_{10}\cdots b_1a_1a_2a_3\cdots a_{52}.$$

Below, we will discuss how the command `.hex()` can show the binary representation of a floating point number in Python.

The case $b_1=\cdots=b_{11}=0$ has a special meaning, corresponding to **subnormal numbers**. In this case, the corresponding number is

$$\pm(0.a_1a_2\cdots a_{52})\cdot 2^{1-1023}=\pm(0.a_1a_2\cdots a_{52})\cdot 2^{-1022}.$$

(The case $a_1 = \cdots = a_{52} = 0$ with a positive sign corresponds to 0, and with a negative sign it corresponds to -0. The floating point representations of 0 and -0 are technically different, despite them being formally equal.) The case $b_1 \cdots b_{11} = 1$ has a special meaning, denoting $\pm\infty$ if $a_i = 0$ for all $1 \leq i \leq 52$, or `NaN` (Not a Number) if $a_i \neq 0$ for some $1 \leq i \leq 52$.

Remark 2.2. A normal number has a unique representation as a double precision floating point number.

Remark 2.3. Here the term "double" signifies that 64 is twice as large as 32. A less precise 32-bit number system, single precision floating point arithmetic, uses a 23 bit mantissa and an 8 bit exponent. Half precision arithmetic, a 16-bit number system uses a 10 bit mantissa and 5 bit exponent. Google's proprietary TPUs also use a 16-bit number system with a 7 bit mantissa and 8 bit exponent, called "bfloat16". NVIDIA GPUs use a 19-bit number system named "tf32" with a 10 bit mantissa (as in half precision) and an 8 bit exponent (as in single precision). Some applications even use 8-bit precision.

The largest exponent of a double precision floating point number is the binary digit with 11 ones (minus 1), minus 1023, i.e.

$$-1023 - 1 + \sum_{i=1}^{11} 2^{i-1} = -1024 + 2^{11} - 1 = -1024 + 2048 - 1 = 1023.$$

The smallest exponent of a double precision floating point normal number is the number 1, minus 1023, i.e. -1022.

So, the largest double precision floating point number is

$$1.1 \cdots 1 \cdot 2^{1023} \approx 1.8 \times 10^{308}.$$

This number in Python is output from the `np.finfo('d').max` command. We can already see some arithmetic issues with this number. For example, `np.finfo('d').max+1` will be equal to `np.finfo('d').max`. Why? (We will discuss this issue more in Section 2.1.) (To see this try the commands `np.finfo('d').max==np.finfo('d').max+1` or `np.finfo('d').max-(np.finfo('d').max+1)`.) Since $2^{1023} < 2^1024$, in some computer algebra systems, 2^{1024} evaluates to infinity. (In Python, positive infinity is represented by `float('inf')`, so expressions such as `2<float('inf')` evaluate to `True`.) In Python 3, there is no memory restriction on the memory storage for an integer. That is, integers (of type `int`) can be represented with more than 64 bits of memory (unlike double precision floating point values, which use only 64 bits of memory). Consequently, `2**1023==2**1023 +1` evaluates to `False`. Also, `2**1030==2**1030 +1` evaluates to `False`. This feature can be nice for certain integer computations, however it can also lead to running out of memory, as in the following program.

```
x = 1
while x != 0:
    x = 2 * x
    print(x)
```

The smallest positive double precision floating point number corresponds to $a_{52} = 1$, and $a_i = 0$ for all $1 \leq i \leq 51$. In this case,

$$0.0 \cdots 01 \cdot 2^{-1022} = 2^{-52} \cdot 2^{-1022} = 2^{-1074} \approx 4.941 \times 10^{-324}.$$

Since this is the smallest positive real number, we might worry about e.g. dividing it by 2. Indeed, `2**(-1074) /2` evaluates to 0, `2**(-1075)` evaluates to 2^{-1074} and `2**(-1076)` evaluates to 0. (Evidently, computing 2^{-1075} results in "rounding up" to the closest double precision number, but $2^{-1074}/2$ results in "rounding down" to the closest double precision number.)

The smallest positive double precision floating point normal number is

$$1 \cdot 2^{1-1023} = 2^{-1022} \approx 2.225 \times 10^{-308}.$$

This number is output from the `np.finfo('d').tiny` command.

As we have seen from a few examples, arithmetic on computers results in rounding errors. Adding small integers to `np.finfo('d').max` results in a rounding error. And dividing 2^{-1074} by two evaluates to zero, another rounding error. The rounding error for additions close to 1 can be approximated by `np.finfo('d').eps`, known as machine epsilon. Machine epsilon is defined to be the distance from 1 to the next largest double precision floating point number, which is

$$2^{-52} \approx 2.22 \times 10^{-16}.$$

Consequently, `(1+2**(-53))-1` evaluates to 0. (We will discuss this issue more in Section 2.1.) Note also that the smallest positive subnormal number is

```
np.finfo('d').tiny*np.finfo('d').eps
```

By default, Python displays the decimal representation of a float point number, rather than its binary representation. However, these decimal representations are perhaps a bit deceiving. For example, 1/10 has an exact, infinite decimal representation as

$$\frac{1}{10} = .0001100110011001\ldots = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + \cdots .$$

Rewriting this in base 16, we have

$$\frac{1}{10} = .0001100110011001\ldots = \frac{1}{2^4}\Big(1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + \cdots\Big).$$

Since these series are infinite, we cannot write 1/10 exactly as a double precision floating point number. We can only write 1/10 approximately as such a number. It turns out that the closest such number is

$$2^{-4}\Big(1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + \cdots + \frac{9}{16^{12}} + \frac{10}{16^{13}}\Big).$$

(Note that 52 binary digits corresponds to $52/4 = 13$ digits in base 16.) Python can display the binary representation of any double precision floating point number with the `float.hex` command. Consecutive groups of four binary digits are rewritten as base sixteen (hexadecimal) digits. In the hexadecimal representation of a number, the letters `a,b,c,d,e,f` correspond to $10, 11, 12, 13, 14, 15$. The hexadecimal representation of 1/10 is then

$$1.999999999999a \cdot 2^{-4}.$$

The command `(1/10).hex()` outputs this representation as

```
0x1.999999999999ap-4
```

As anticipated, the remaining thirteen digits `999999999999a` represent the mantissa of the hexadecimal representation of 1/10 described above.

This rounding error can propagate through other operations. For example, `.3/.1` does not evaluate to 3, since the numerator is slightly smaller than .3 and the denominator is slightly larger than .1. In order to see that `.3/.1`, we can either evaluate `.3/.1 <3` or check the hexadecimal representation of `.3/.1` in Python. Here are some examples of exact hexadecimal representations of numbers:

Real Number	Python Command	Hex Representation
2^{-1074}	`np.finfo('d').tiny * np.finfo('d').eps`	`0.0000000000001p-1022`
2^{-1022}	`np.finfo('d').tiny`	`1.0000000000000p-1022`
2^{-52}	`np.finfo('d').eps`	`1.0000000000000p-52`
-2^{-52}	`-np.finfo('d').eps`	`-1.0000000000000p-52`
$(2-2^{-52})\cdot 2^{1023}$	`np.finfo('d').max`	`1.fffffffffffffp+1023`
0	0.0	`0.0p+0`
0	-0.0	`-0.0p+0`

Exercise 2.4. Let $\mathcal{F}$ be the set of all positive double precision floating point numbers (except for `NaN`s and `Inf`s), that have the exponent `1023` (in their hexadecimal representation in Python). (For example, `np.finfo('d').max` is in $\mathcal{F}$, since)

- How many elements are in $\mathcal{F}$? That is, what is the cardinality $|\mathcal{F}|$ of $\mathcal{F}$.
- What fraction of elements of $\mathcal{F}$ are in the interval $[2^{1023}, 2^{1024})$?
- What fraction of elements of $\mathcal{F}$ are in the interval $[2^{1023}, \frac{3}{2}2^{1023})$?
- Using e.g. Numpy's `random` function, write a program that estimates the fraction of $x \in \mathcal{F}$ that satisfy the expression `x * (1/x)==1`. (It would take a pretty long time to check how many elements of $\mathcal{F}$ satisfy this equation, so you should not do that.)

Warning: Numpy's `random` function tries to find a uniformly random chosen number in the interval $(0, 1)$ and then round it to the nearest floating point number. This operation is different than choosing a floating point number uniformly over all (positive) floating point numbers with a fixed exponent. (This is the point of the second and third items of this exercise.) For this reason, your answer to the last part of the question might be different from the output of the program:

```
x = np.random.rand(1000)
sum(x* (1/x) == 1) / 1000
```

2.1. Floating Point Arithmetic.

Definition 2.5 (Floating Point Addition). Let x, y be positive normal numbers, as defined in Definition 2.1. Then the addition of x and y is defined as follows.

- Represent each of x and y as binary numbers of the form
$$x = (1.a_1a_2\cdots a_{52})\cdot 2^{e_x}, \qquad y = (1.\widetilde{a}_1\widetilde{a}_2\cdots\widetilde{a}_{52})\cdot 2^{e_y}.$$
(Here e_x, e_y are integers and $a_1, \ldots, a_{52}, \widetilde{a}_1, \ldots, \widetilde{a}_{52} \in \{0, 1\}$.)
- Write both x and y using the same exponent. For example, if $e_x \geq e_y$, we write
$$x = (1.a_1a_2\cdots a_{52})\cdot 2^{e_x}, \qquad y = (.0\cdots 01\widetilde{a}_1\widetilde{a}_2\cdots\widetilde{a}_{52})\cdot 2^{e_x}.$$
- Add the digits x and y componentwise with carrying rules. (Since the numbers have the same exponent, you can use the addition and carry rules you learned in grade school.) We then get
$$x + y = (1.c_1\cdots c_k)\cdot 2^{e_x},$$

for some positive integer $k \geq 52$. (In the case $e_x = e_y$, we might need to change the exponent in this step to write $x + y$ itself as a floating point number.)

- In the case $k > 52$, round the result from the previous step to a floating point number such as
$$(1.c_1 \cdots c_{52}) \cdot 2^{e_x}.$$
(Truncating to 52 decimal places corresponds to "rounding down.") (Python will round to the nearest floating point number, and it will round towards zero in case of a tie. For example `1+eps/2` returns 1, `1+eps/1.9` is equal to `1+eps`, and `-1-(eps/2)` returns -1.)

(According to the above definition, we might need to take k very large in order to perform addition. However, Python only requires $k \leq 55$ bits to store y during the addition step.)

Example 2.6. Suppose for simplicity we have a floating point arithmetic system in base ten with three digits stored in the mantissa, and we want to add
$$x = 1.312 \times 10^3, \qquad y = 1.929 \times 10^2.$$
We first write
$$x = 1.312 \times 10^3, \qquad y = .1929 \times 10^3.$$
Adding componentwise leads to
$$x + y = 1.5049 \times 10^3.$$
Since the arithmetic system only stores three digits, the final computed value of $x+y$ is the rounded answer
$$1.505 \times 10^3.$$

In certain implementations, extra unnecessary bits might be discarded in the computation described in Definition 2.5, e.g. adding $x = 2^{50}$ and $y = 2^{-50}$ naïvely might require about 100 bits of storage for y when we write both x and y using the same exponent, but such storage is not really needed since the addition of x and y is just x. (In this case, adding y to x does not change the digits of x at all.)

Remark 2.7. Floating point subtraction is defined in a similar way to addition. Multiplication and division are even simpler, since Step 2 of Definition 2.5 is not needed. For example, to multiply, just multiply the mantissas and add the exponents.

Proposition 2.8. *Let x, y be positive normal numbers, as defined in Definition 2.1. Assume that $x + y < 2^{1024}$. Let $\mathrm{fl}(x + y)$ denote the double precision floating point representation of $x + y$. Then there exists $\delta \in \mathbb{R}$ with $|\delta| \leq 2^{-52}$ such that*
$$\mathrm{fl}(x + y) = (x + y)(1 + \delta).$$

Proof. This follows from the last part of Definition 2.5. □

Definition 2.9. Let x be a real number, and let $x^* \in \mathbb{R}$ be a computed value of x (such as $\mathrm{fl}(x)$). We define the **absolute error** of x^* to be
$$|x - x^*|.$$
If $x \neq 0$, we define the **relative error** of x^* to be
$$\frac{|x - x^*|}{|x|}$$

So, Proposition 2.8 says that the relative error of the computation $\mathrm{fl}(x+y)$ relative to $x+y$ is

$$\frac{|\mathrm{fl}(x+y)-(x+y)|}{x+y}\leq 2^{-52},$$

whenever $x, y > 0$ are normal double precision floating number numbers with $x+y < 2^{1024}$. Iterating Proposition 2.8 k times gives

Proposition 2.10. *Let $x_1, \ldots, x_k$ be positive normal numbers, as defined in Definition 2.1. Assume that $\sum_{i=1}^k x_i < 2^{1024}$. Then the relative error of $\mathrm{fl}\Big(\sum_{i=1}^k x_i\Big)$ relative to $\sum_{i=1}^k x_i$ is at most*

$$(1+2^{-52})^{k-1}-1 \approx (k-1)2^{-52}.$$

To justify the approximation, note that the binomial theorem implies that

$$(1+2^{-52})^{k-1}=\sum_{j=0}^{k-1}\binom{k-1}{j}(2^{-52})^j = 1+(k-1)2^{-52}+(1/2)(k-1)(k-2)(2^{-52})^2+\cdots$$

If k is small (say $k < 2^{20}$), then the term $(1/2)(k-1)(k-2)(2^{-52})^2$ is much smaller than $(k-1)2^{-52}$. The same comment applies for the remaining terms in the sum.

Exercise 2.11. Do the following plot in Python

```
import matplotlib.pyplot as plt
x = np.arange(.988, 1.012, .0001)
y = x**7 - 7*x**6 + 21*x**5 - 35*x**4 + 35*x**3 - 21*x**2 + 7*x - 1
plt.plot(x, y)
plt.show()
```

This is the function $y(x) = (x-1)^7$ for $x \in [.988, 1.012]$. Does the plot look like a polynomial? Explain why or why not.

Exercise 2.12. Suppose we want to solve the linear system of equations

$$\begin{aligned}17x_1 + 5x_2 &= 22,\\ 1.7x_1 + .5x_2 &= 2.2.\end{aligned}$$

Note that $(x_1, x_2) = (1, 1)$ is a solution to this system of equations.

Python can numerically solve this system with the following program

```
A = np.array([ [ 17, 5], [1.7, .5] ])
b = np.array([22, 2.2])
x = np.linalg.solve(A, b)
```

- What is the solution x that is output from the program?
- Is the output of the program an actual solution of the original system of equations?
- What is the determinant of A? What does Python output from the command `np.linalg.det(A)`?

Warning: for a 2×2 matrix A and a scalar $t > 0$, we have $\det(tA) = t^2\det(A)$. So, the value of a determinant does not necessarily say anything about how well we can solve a linear system of equations of the form $Ax = b$.

Exercise 2.13. The sin function, like other special functions such as $\cos, \exp, \log$, etc., cannot be computed exactly on a computer. A common way to compute these special functions is via power series. Recall that sin has the following power series that is absolutely convergent for all $x \in \mathbb{R}$:

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

With this power series in mind, run the following program when $x = \pi/2, 11\pi/2, 21\pi/2$ and $31\pi/2$. (Before you run the program, set x to a specific value.)

```
s = 0
t = x
n = 1
while s + t != s:
    s = s + t
    t = -(x**2) * t / ((n + 1) * (n + 2))
    n = n + 2
print(s)
```

When the program terminates, the value of s is the computed value of $\sin(x)$. For each value of x stated above, answer the following:

- What is the absolute error of the computation of $\sin(x)$?
- How many terms of the power series were used in the computation of $\sin(x)$?
- What is the largest term in the power series expansion of $\sin(x)$? (Hint: consider using the `numpy.max` command)

2.1.1. *Subtraction and Loss of Significance.* Analogues of Propositions 2.8 and 2.10 hold for multiplication and division. For example

Proposition 2.14. *Let x, y be positive normal numbers, as defined in Definition 2.1. Let $\odot$ denote multiplication or division. Assume that $2^{-1022} < x \odot y < 2^{1024}$. Let $\mathrm{fl}(x \odot y)$ denote the double precision floating point representation of $x \odot y$. Then there exists $\delta \in \mathbb{R}$ with $|\delta| \leq 2^{-52}$ such that*

$$\mathrm{fl}(x \odot y) = (x \odot y)(1 + \delta).$$

Unfortunately Proposition 2.10 can not hold for a succession of addition and subtraction. To see why, suppose for simplicity we have a floating point arithmetic system in base ten with three digits stored in the mantissa, and we want to subtract

$$x = 1.234 \times 10^0, \qquad \text{and} \qquad y = 1.233 \times 10^0.$$

When we perform the subtraction $x - y$, we get

$$0.001 \times 10^0$$

The final answer must be a floating point number, so the output of the program is

$$1.000 \times 10^{-3}.$$

Since x and y shared the most significant digits in their mantissas, the subtraction $x - y$ had only one significant digit. Then the returned value of $x - y$ has zero significant digits in the mantissa. Since significant digits were lost in the mantissa, this issue is known as a **loss of significance**.

For this single subtraction, no error has actually occurred, since $x - y = 10^{-3}$. However, combining other operations with subtractions can cause substantial errors. For example, the expression

$$(1 + 2^{-53}) - 1$$

returns the value 0 in Python, when it should be equal to 2^{-53}. The absolute error of 2^{-53} is quite small, but the relative error is not even defined. Even more alarming, the expression

$$2^{53}((1 + 2^{-53}) - 1)$$

returns the value 1 in Python, when it should be equal to 0. That is, there is an absolute error of 1. This observation leads to the following heuristic.

Heuristic for Floating Point Arithmetic: Subtractions are dangerous, but addition, multiplication and division are generally safe (concerning relative errors).

The relative safety of addition, multiplication and division follows from the analogues of Propositions 2.8 and 2.10. The danger of using subtraction can be formalized in the following statement.

Proposition 2.15. *Let x and y be positive normal double precision floating point numbers with $x > y$. (Since $x > y$, $1 - y/x > 0$.) Let p, q be nonnegative integers such that*

$$2^{-q} \le 1 - \frac{y}{x} \le 2^{-p}.$$

Let d be the number of zeros at the end of the mantissa in the computation of $x - y$. (We could say d is the number of significant binary digits that are lost in computing $x - y$.) Then

$$p \le d \le q.$$

Proof. Let $1 \le s, t < 2$ and let m, n be integers such that

$$x = s2^m, \qquad y = t2^n.$$

Write $y = t2^{n-m}2^m$ so that $x - y = (s - t2^{n-m})2^m$. Since $x > y$, $s - t2^{n-m} > 0$, so that $s - t2^{n-m}$ is a mantissa representation of $x - y$. This mantissa satisfies

$$s - t2^{n-m} = s\Big(1 - \frac{t2^n}{s2^m}\Big) = s\Big(1 - \frac{y}{x}\Big).$$

Since $1 \le s < 2$, our assumption implies that

$$2^{-q} \le s - t2^{n-m} < 2 \cdot 2^{-p}.$$

That is, the mantissa's binary representation starts with at least p zeros, and at most q zeros. □

Example 2.16. Near zero, the function

$$f(x) = \sqrt{x^2 + 1} - 1$$

exhibits some loss of significance errors, as the following plot shows.

```
import matplotlib.pyplot as plt
x = np.arange(-10 ** - 7, 10 ** -7, 10 ** -10)
y = np.sqrt( x**2 + 1) - 1
z = x**2 / (np.sqrt(2**2 + 1) + 1)
plt.plot(x, y, 'r', x, z, 'b')
plt.text(-.8e-7, 4e-15, r'$\sqrt{x^{2}+1}-1$', color='red')
plt.text(.5e-7, .5e-15, r'$\frac{x^{2}}{1+\sqrt{x^{2}+1}}$', color='blue')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.show()
```

However, by multiplying and dividing by $\sqrt{x^2+1}+1$, we can rewrite f as

$$f(x) = \frac{x^2}{\sqrt{x^2+1}+1}, \qquad \forall\, x \in \mathbb{R},$$

which avoids the loss of significance of the previous formula.

Exercise 2.17. Suppose we want to compute the quantity

$$x - \sin(x)$$

for any real $x \in \mathbb{R}$. For x near zero, there will be a loss of significance error, so we should perhaps try to find a better way to compute this quantity.

- Find the loss of significance (i.e. the number of zero bits at the end of the binary mantissa) when $x - \sin(x)$ is computed directly in double precision floating point arithmetic in Python, when $x = 2^{-25}$.
- Find the loss of significance (i.e. the number of zero bits at the end of the mantissa) when $x - \sin(x)$ is computed as
$$\frac{x^3}{3!} - \frac{x^5}{5!},$$
when $x = 2^{-25}$. (Your answer can be off by one or two from the true value.)
- Estimate the relative error when $x - \sin(x)$ is computed as
$$\frac{x^3}{3!} - \frac{x^5}{5!},$$
when $x = 2^{-25}$. (Your answer does not have to exactly correct. It is okay to be approximately correct.)

Exercise 2.18. This exercise examines an unstable recurrence computation.

Consider the following recursion with $x_0 := 1$ and $x_1 := 1/3$.

$$x_{n+1} = \frac{13}{3}x_n - \frac{4}{3}x_{n-1}, \qquad \forall\, n \geq 1.$$

- Verify that the recurrence is solved by $x_n := (1/3)^n$ for all $n \geq 0$.
- Using Python, solve for x_{40}. For example, use
```
x = np.array([1, 1/3])
for i in range(2,41):
    x = np.append(x, (13/3)*x[i-1]  - (4/3)*x[i-2])
print(x[40])
```
Is the answer what you expected to get? (Hint: examine a logarithmically scaled plot in the y-axis, using `matplotlib.pyplot.semilogy`.)
- With a different initial condition, the above recurrence can have other solutions. To find them, rewrite the recurrence as
$$\begin{pmatrix} 13/3 & -4/3 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_n \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} x_{n+1} \\ x_n \end{pmatrix}, \qquad \forall\, n \geq 1.$$

Then note that the eigenvalues of the matrix $A := \begin{pmatrix} 13/3 & -4/3 \\ 1 & 0 \end{pmatrix}$ are 1/3 and 4, so iterating the recurrence shows that

$$\begin{pmatrix} 13/3 & -4/3 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} x_1 \\ x_0 \end{pmatrix} = \begin{pmatrix} x_{n+1} \\ x_n \end{pmatrix}, \qquad \forall\, n \geq 0.$$

Since A has two distinct eigenvalues, it is diagonalizable, so if $\begin{pmatrix} x_2 \\ x_1 \end{pmatrix}$ is written as a linear combination $a_1 v_1 + a_2 v_2$ of the corresponding eigenvectors $v_1, v_2 \in \mathbb{R}^2$ of A, the recurrence becomes

$$a_1 (1/3)^{n0} v_1 + a_2 4^n v_2 = \begin{pmatrix} x_{n+1} \\ x_n \end{pmatrix}, \qquad \forall\, n \geq 0.$$

- Show that in the case $x_1 = 1$ and $x_2 = 1/3$, we have $a_2 = 0$. However, small numerical errors that occur in the computation of the recurrence correspond to a_2 being computed to be nonzero. Explain how this relates to the logarithmic plot you examined above.

2.1.2. *Simulation of Random Variables.*

Remark 2.19. A Monte Carlo simulation simulates a large number samples from some random quantity. For example, the command `rng=np.random.rand(1000)` generates a length 1000 vector that simulates 1000 numbers that are equally likely to take any value in $(0, 1)$. And the command `np.random.randint(0, 2, 1000)` represents a vector of 1000 fair coin flips, since each entry of the vector should have probability 1/2 of taking the value 0 or 1. (The same is accomplished with `np.round(np.random.rand(1000))` .)

In a Monte Carlo simulation, one often sums the results of n samples and then divides by n. For example, the Law of Large Numbers says that, for a large number (such as 10000), `np.mean(np.random.randint(0, 2, 1000))` should be close to 1/2. That is, roughly half of 10000 coin flips will be heads, and roughly half of these flips will be tails. (Though actually it is unlikely that exactly 5000 of the coin flips will be heads.)

Exercise 2.20 (Numerical Integration)**.** Consider the function

$$f(t) := t^3 + 1.$$

In this case, we can easily compute

$$\int_0^1 f(t)dt = \frac{5}{4}.$$

Sometimes, especially in computer graphics applications, integrals are too complicated to compute directly, so we instead use randomness to estimate the integral. That is, we pick n random points in $[0, 1]$, and average the values of f at these points, as in the following program.

```
n = 10**5
t = np.random.rand(n)
f = t**3 +1
np.mean(f)
```

Using this program with $n = 10^5, 10^6, 10^7$ and 10^8, report the estimated values for the integral of f, along with their relative errors.

Now, compute the exact value of $\int_3^5 \log x dx$, and modify the above program to give estimates for the value of this integral and report relative errors, using a number of points n where $n = 10^5, 10^6, 10^7$ and 10^8.

Remark 2.21. When Python or other computer programs generate "random numbers" using e.g. `np.random.rand` or `np.random.randn`, these numbers are not actually random or independent. These numbers are **pseudorandom**. That is, functions such as `rand` output numbers in a deterministic way, but these numbers behave as if they were random. All "random" numbers generated by computers are actually pseudorandom, and this includes slot machines at casinos, video games, etc. So, when using Monte Carlo simulation as we did above, we should be careful about interpreting our results, since it is generally impossible to take random samples on a computer.

And, theoretically, if you knew enough about the random number generator that a slot machine is using, you could predict its output.

2.1.3. *Statistical Estimation and Numpy.*

Exercise 2.22. In this exercise, we will compare the run time of built-in vectorized functions versus a naive for loop

- Compare the time it takes to compute a dot product using numpy's `np.dot` function, versus using a for loop. More specifically, use `x=np.random.randn(10 ** k)` and `y=np.random.randn(10 ** k)` for $k = 3, 4, 5, 6, 7$, and compute the dot product of x and y in the two different specified ways (vectorized numpy and for loop). Do the run times increase exponentially with k?
- Compare the time it takes to compute a matrix product using numpy's `np.dot` function, versus using a for loop. More specifically, use `A=np.random.randn(10 ** k, 10 ** k)` and `B=np.random.rand(10 ** k, 10 ** k)` for $k = 1, 2, 3, 4$, and compute the matrix product AB using `A @ B`, versus a for loop. Do the run times increase exponentially with k?

(Optional:) Repeat the above exercise for 32-bit arithmetic, using the `np.float32` command.

Exercise 2.23. The links below contain `.csv` files, each with 1000 (pseudo) random samples from a Gaussian distribution with variance one and unknown mean $\mu \in \mathbb{R}$

gaussian data

gaussian data v2

Recall that a basic question in parametric statistics is to estimate the unknown mean μ. From statistics class, we know that a good estimator for the mean will be the sample mean (since e.g. it is the MLE for the mean). Using the following commands, we can import the first `.csv` file into a Numpy array, and then take the sample mean to estimate μ.

```
x = np.genfromtxt("gaussian_data.csv", delimiter=",")
np.mean(x)
```

The output is $-.00968$. I used $\mu = 0$ to generate these samples, so the mean estimate is pretty close to reality. However, the second file is exactly the same as the first, but I intentionally created two outliers to skew the final result. The output of the above program for the second file is 11371.66, which is quite far from the true value $\mu = 0$. With this example in mind, we ask: what is a good estimate of the unknown mean μ that is robust to noise (or robust to outliers)? There are many possible good answers, and one such answer is the median. The following program

```
x = np.genfromtxt("gaussian_datav2.csv", delimiter=",")
np.median(x)
```

has output $-.03$. Can you think of a better way to remove the outliers and estimate the unknown mean? (This question is intentionally open ended.)

2.1.4. *Additional Comments.* Classical numerical analysis often bounds the numerical errors of a numerical algorithm, e.g. that estimates the integral of a function.

Modern numerical analysis also examines the behavior of algorithms that work well with noisy data (i.e. algorithms that work well in the average case, rather than the worst case). Sometimes we can even guarantee the performance of an algorithm when it is given adversarial data (i.e. some inputs to the algorithm can be chosen arbitrarily).

For more details on the use of "guard bits" and "sticky bits" in implementation of e.g. addition in Python, see Numerical Computing with IEEE Floating Point Arithmetic, Michael Overton

3. Numerical Linear Algebra

3.0.1. *Review of Linear Algebra.*

Definition 3.1 (**Linear combination**). Let V be a vector space over a field $\mathbb{F}$. Let $u_1, \ldots, u_n \in V$ and let $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$. Then $\sum_{i=1}^n \alpha_i u_i$ is called a **linear combination** of the vector elements $u_1, \ldots, u_n$.

Definition 3.2 (**Linear dependence**). Let V be a vector space over a field $\mathbb{F}$. Let S be a subset of V. We say that S is **linearly dependent** if there exists a finite set of vectors $u_1, \ldots, u_n \in S$ and there exist $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$ which are not all zero such that $\sum_{i=1}^n \alpha_i u_i = 0$.

Definition 3.3 (**Linear independence**). Let V be a vector space over a field $\mathbb{F}$. Let S be a subset of V. We say that S is **linearly independent** if S is not linearly dependent.

Example 3.4. The set $S = \{(1,0),(0,1)\}$ is linearly independent in $\mathbb{R}^2$. The set $S \cup (1,1)$ is linearly dependent in $\mathbb{R}^2$, since $(1,0)+(0,1)-(1,1)=0$.

Definition 3.5 (**Span**). Let V be a vector space over a field $\mathbb{F}$. Let $S \subseteq V$ be a finite or infinite set. Then the **span** of S, denoted by $\mathrm{span}(S)$, is the set of all finite linear combinations of vectors in S. That is,

$$\mathrm{span}(S) = \left\{ \sum_{i=1}^n \alpha_i u_i \colon n \in \mathbb{N}, \alpha_i \in \mathbb{F}, u_i \in S, \forall\, i \in \{1, \ldots, n\} \right\}.$$

Remark 3.6. We define $\mathrm{span}(\emptyset) := \{0\}$.

Theorem 3.7 (**Span as a Subspace**). *Let V be a vector space over a field $\mathbb{F}$. Let $S \subseteq V$. Then $\mathrm{span}(S)$ is a subspace of V such that $S \subseteq \mathrm{span}(S)$. Also, any subspace of V that contains S must also contain $\mathrm{span}(S)$.*

Definition 3.8 (**Normed Linear Space**). Let $\mathbb{F}$ denote either $\mathbb{R}$ or $\mathbb{C}$. Let V be a vector space over $\mathbb{F}$. A **normed linear space** is a vector space V equipped with a **norm**. A norm is a function $V \to \mathbb{R}$, denoted by $\|\cdot\|$, which satisfies the following properties.

(a) For all $v \in V$, for all $\alpha \in \mathbb{F}$, $\|\alpha v\| = |\alpha| \, \|v\|$. (Homogeneity)
(b) For all $v \in V$ with $v \neq 0$, $\|v\|$ is a positive real number; $\|v\| > 0$. And $v = 0$ if and only if $\|v\| = 0$. (Positive definiteness)
(c) For all $v, w \in V$, $\|v + w\| \leq \|v\| + \|w\|$. (Triangle Inequality)

Definition 3.9 (**Complex Conjugate**)**.** Let $i := \sqrt{-1}$. Let $x, y \in \mathbb{R}$, and let $z = x+iy \in \mathbb{C}$. Define $\overline{z} := x - iy$. Define $|z| := \sqrt{x^2 + y^2}$. Note that $|z|^2 = z\overline{z}$.

Definition 3.10 (**Inner Product**)**.** Let $\mathbb{F}$ denote either $\mathbb{R}$ or $\mathbb{C}$. Let V be a vector space over $\mathbb{F}$. An **inner product space** is a vector space V equipped with an **inner product**. An inner product is a function $V \times V \to \mathbb{F}$, denoted by $\langle \cdot, \cdot \rangle$, which satisfies the following properties.

(a) For all $v, v', w \in V$, $\langle v + v', w \rangle = \langle v, w \rangle + \langle v', w \rangle$. (Linearity in the first argument).
(b) For all $v, w \in V$, for all $\alpha \in \mathbb{F}$, $\langle \alpha v, w \rangle = \alpha \langle v, w \rangle$. (Homogeneity in the first argument)
(c) For all $v \in V$, if $v \neq 0$, then $\langle v, v \rangle$ is a positive real number; $\langle v, v \rangle > 0$. (Positivity)
(d) For all $v, w \in V$, $\langle v, w \rangle = \overline{\langle w, v \rangle}$. (Conjugate symmetry)

Exercise 3.11. Using the above properties, show the following things.

(e) For all $v, v', w \in V$, $\langle w, v + v' \rangle = \langle w, v \rangle + \langle w, v' \rangle$. (Linearity in the second argument)
(f) For all $v, w \in V$, for all $\alpha \in \mathbb{F}$, $\langle v, \alpha w \rangle = \overline{\alpha} \langle v, w \rangle$.
(g) For all $v \in V$, $\langle v, 0 \rangle = \langle 0, v \rangle = 0$.
(h) $\langle v, v \rangle = 0$ if and only if $v = 0$.

Remark 3.12. If $\mathbb{F} = \mathbb{R}$, then property (d) says that $\langle v, w \rangle = \langle w, v \rangle$.

Lemma 3.13. *Let $\langle , \rangle$ be an inner product on a vector space V. Then the function $\|\cdot\| : V \to \mathbb{R}$ defined by $\|v\| := \sqrt{\langle v, v \rangle}$ is a norm on V.*

Definition 3.14 (**Orthogonal Vectors**)**.** Let V be an inner product space, and let $v, w \in V$. We say that v, w are **orthogonal** if $\langle v, w \rangle = 0$.

Definition 3.15 (**Orthogonal Set, Orthonormal Set**)**.** Let V be an inner product space and let $(v_1, \ldots, v_n)$ be a collection of vectors in V. The set of vectors $(v_1, \ldots, v_n)$ is said to be **orthogonal** if $\langle v_i, v_j \rangle = 0$ for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$. If additionally $\langle v_i, v_i \rangle = 1$ for all $i \in \{1, \ldots, n\}$, the set of vectors $(v_1, \ldots, v_n)$ is called **orthonormal**.

Corollary 3.16. *Let V be an inner product space, and let $v_1, \ldots, v_n \in V$ be an orthonormal set of vectors. Then*

$$\left\| \sum_{i=1}^{n} \alpha_i v_i \right\|^2 = \sum_{i=1}^{n} |\alpha_i|^2 .$$

Corollary 3.17. *Any set of orthonormal vectors is linearly independent.*

Definition 3.18 (**Orthonormal Basis**)**.** Let V be an inner product space. An **orthonormal basis** of V is a collection $(v_1, \ldots, v_n)$ of orthonormal vectors that is also a basis for V.

Corollary 3.19. *Let V be an n-dimensional inner product space. Let $(v_1, \ldots, v_n)$ be an orthonormal set in V. Then $(v_1, \ldots, v_n)$ is an orthonormal basis of V.*

Theorem 3.20. *Let V be an inner product space. Let $(v_1, \ldots, v_n)$ be an orthonormal basis of V. Then, for any $v \in V$, we have*

$$v = \sum_{i=1}^{n} \langle v, v_i \rangle v_i.$$

Definition 3.21 (Unit Vector). Let V be a normed linear space, and let $v \in V$. If $\|v\| = 1$, we say that v is a **unit vector**.

Remark 3.22. Let $v \neq 0$. Then $v / \|v\|$ is a unit vector.

Theorem 3.23 (Gram-Schmidt Orthogonalization). *Let $v_1, \ldots, v_n$ be a linearly independent set of vectors in an inner product space V. Then we can create an orthogonal set of vectors in V as follows. Define*

$$\begin{aligned} w_1 &:= v_1. \\ w_2 &:= v_2 - \left\langle v_2, \frac{w_1}{\|w_1\|} \right\rangle \frac{w_1}{\|w_1\|}. \\ w_3 &:= v_3 - \left\langle v_3, \frac{w_1}{\|w_1\|} \right\rangle \frac{w_1}{\|w_1\|} - \left\langle v_3, \frac{w_2}{\|w_2\|} \right\rangle \frac{w_2}{\|w_2\|}. \end{aligned}$$

And so on. In general, for $k \in \{2, \ldots, n\}$, define

$$w_k := v_k - \sum_{j=1}^{k-1} \left\langle v_k, \frac{w_j}{\|w_j\|} \right\rangle \frac{w_j}{\|w_j\|}.$$

Then for each $k \in \{1, \ldots, n\}$, $(w_1, \ldots, w_k)$ is an orthogonal set of nonzero vectors in V. Also, $\mathrm{span}(w_1, \ldots, w_k) = \mathrm{span}(v_1, \ldots, v_k)$ for each $k \in \{1, \ldots, n\}$. Finally, note that the set $(w_1/\|w_1\|, \ldots, w_n/\|w_n\|)$ is an orthonormal set of vectors in V with the same span as $v_1, \ldots, v_n$.

Definition 3.24 (Transpose). Let A be an $m \times n$ matrix with entries A_{ij}, $1 \leq i \leq m$, $1 \leq j \leq n$. Then the **transpose** A^T of A is defined to be the $n \times m$ matrix with entries $(A^T)_{ij} := A_{ji}$, $1 \leq i \leq n$, $1 \leq j \leq m$.

Exercise 3.25. Let A be an $m \times n$ matrix. Let B be an $\ell \times m$ matrix. Show that $(BA)^T = A^T B^T$.

Remark 3.26. If A is an $n \times n$ invertible matrix, then $I_n^T = (AA^{-1})^T = (A^{-1})^T A^T$, so A^T is also invertible.

Definition 3.27 (Adjoint of a Matrix). Let A be an $m \times n$ matrix with $A_{jk} \in \mathbb{C}$, $1 \leq j \leq m$, $1 \leq k \leq n$. The **adjoint** of A, denoted by A^*, is an $n \times m$ matrix with entries $(A^*)_{jk} := \overline{A_{kj}}$, $1 \leq j \leq n$, $1 \leq k \leq m$.

Definition 3.28 (Normal Matrix). Let A be an $n \times n$ matrix with values in $\mathbb{C}$. We say that A is **normal** if $AA^* = A^*A$.

Definition 3.29 (Self-Adjoint Matrix). Let $\mathbb{F}$ denote $\mathbb{R}$ or $\mathbb{C}$. A square matrix A with elements in $\mathbb{F}$ is said to be **self-adjoint** if $A = A^*$. The term **Hermitian** is a synonym of self-adjoint.

Definition 3.30 (**Unitary Matrix/ Orthogonal Matrix**). Let A be an $n \times n$ matrix with elements in $\mathbb{C}$. We say that A is **unitary** if $AA^* = A^*A = I$. In the case that A has real entries and $AA^T = A^TA = I$, we say that A is **orthogonal**.

Definition 3.31. A matrix A is said to be **diagonalizable** if there exists an invertible matrix Q and a diagonal matrix D such that $A = QDQ^{-1}$. That is, a matrix A is diagonalizable if and only if it is similar to a diagonal matrix.

Definition 3.32 (**Eigenvector and Eigenvalue**). Let V be a vector space over a field $\mathbb{F}$. Let $T \colon V \to V$ be a linear transformation. An **eigenvector** of T is a nonzero vector $v \in V$ such that, there exists $\lambda \in \mathbb{F}$ with $T(v) = \lambda v$. The scalar λ is then referred to as the **eigenvalue** of the eigenvector v.

Theorem 3.33 (**The Spectral Theorem for Normal Matrices**). *Let $\mathbb{F}$ denote either $\mathbb{R}$ or $\mathbb{C}$. Let A be an $n \times n$ matrix with entries in $\mathbb{F}$. Then there exists an orthonormal basis of $\mathbb{F}^n$ consisting of eigenvectors of A. In particular, A is diagonalizable with $A = QDQ^{-1}$, where the columns of Q are eigenvectors of A and $QQ^* = Q^*Q = I$.*

Theorem 3.34 (**The Spectral Theorem for Self-Adjoint Matrices**). *Let $\mathbb{F}$ denote either $\mathbb{R}$ or $\mathbb{C}$. Let A be an $n \times n$ matrix with entries in $\mathbb{F}$. Then there exists an orthonormal basis of F^n consisting of eigenvectors of A. In particular, A is diagonalizable with $A = QDQ^{-1}$, where the columns of Q are eigenvectors of A and $QQ^* = Q^*Q = I$. Moreover, all eigenvalues of A are real, i.e. D has real entries.*

Theorem 3.35 (**The Spectral Theorem for Unitary Matrices**). *Let $\mathbb{F}$ denote either $\mathbb{R}$ or $\mathbb{C}$. Let A be an $n \times n$ matrix with entries in $\mathbb{F}$. Then there exists an orthonormal basis of F^n consisting of eigenvectors of A. In particular, A is diagonalizable with $A = QDQ^{-1}$, where the columns of Q are eigenvectors of A. Moreover, all eigenvalues of T have absolute value 1.*

We will discuss algorithms for Spectral Theorems in Section 3.2.3.

3.1. Row Operations, Multiplication, Gaussian Elimination, LU, Ax=b.

We begin our discussion of row operations on matrices with some examples.

Example 3.36 (**Type 1: Interchange two Rows**). For example, we can swap the first and third rows of the matrix

$$\begin{pmatrix} 1 & 2 \\ 3 & 5 \\ 0 & 8 \end{pmatrix}$$

to get

$$\begin{pmatrix} 0 & 8 \\ 3 & 5 \\ 1 & 2 \end{pmatrix}.$$

Define

$$E := \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

Note that

$$E\begin{pmatrix}1 & 2\\ 3 & 5\\ 0 & 8\end{pmatrix} = \begin{pmatrix}0 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 0\end{pmatrix}\begin{pmatrix}1 & 2\\ 3 & 5\\ 0 & 8\end{pmatrix} = \begin{pmatrix}0 & 8\\ 3 & 5\\ 1 & 2\end{pmatrix}.$$

Remark 3.37. E as defined above is invertible. In fact, $E = E^{-1}$. In general, if E is the $n \times n$ matrix that swaps two rows of an $n \times n$ matrix A, then EA is A with those two rows swapped. So $EEA = A$ for all $n \times n$ matrices A, so $EE = I_n$, i.e. E is invertible.

Example 3.38 (**Type 2: Multiply a row by a nonzero scalar**)**.** For example, let's multiply the second row of the following matrix by 2.

$$\begin{pmatrix}1 & 2\\ 3 & 5\\ 0 & 8\end{pmatrix}.$$

We then get

$$\begin{pmatrix}1 & 2\\ 6 & 10\\ 0 & 8\end{pmatrix}.$$

Define

$$E := \begin{pmatrix}1 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 1\end{pmatrix}.$$

Note that

$$E\begin{pmatrix}1 & 2\\ 3 & 5\\ 0 & 8\end{pmatrix} = \begin{pmatrix}1 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 2\\ 3 & 5\\ 0 & 8\end{pmatrix} = \begin{pmatrix}1 & 2\\ 6 & 10\\ 0 & 8\end{pmatrix}$$

Remark 3.39. E as defined above has inverse

$$\begin{pmatrix}1 & 0 & 0\\ 0 & 1/2 & 0\\ 0 & 0 & 1\end{pmatrix}.$$

In general, suppose E corresponds to multiplying the i^{th} row of a given matrix by $\alpha \in \mathbb{F}$, $\alpha \neq 0$. Then E is a matrix with ones on the diagonal, except for the i^{th} entry on the diagonal, which is α. And all other entries of E are zero. Then, we see that E^{-1} exists and is a matrix with ones on the diagonal, except for the i^{th} entry on the diagonal, which is α^{-1}. And all other entries of E^{-1} are zero. In particular, E is invertible.

Example 3.40 (**Adding one row to another**)**.** Let's add two copies of the first row of the following matrix to the third row.

$$\begin{pmatrix}1 & 2\\ 3 & 5\\ 0 & 8\end{pmatrix}.$$

We then get

$$\begin{pmatrix}1 & 2\\ 3 & 5\\ 2 & 12\end{pmatrix}.$$

Define
$$E := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix}.$$
Note that
$$E \begin{pmatrix} 1 & 2 \\ 3 & 5 \\ 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 5 \\ 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 5 \\ 2 & 12 \end{pmatrix}.$$

Remark 3.41. E as defined above has inverse
$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}.$$
That is, adding 2 copies of row one to row three is inverted by adding -2 copies of row one to row three. In a similar way, a general row addition operator is seen to be invertible.

Remark 3.42 (Summary of Row Operations). The three row operations (Type 1, Type 2, and Type 3) are all invertible.

Remark 3.43 (Solving Systems of Linear Equations). Let A be an $m \times n$ matrix, let $x \in \mathbb{R}^n$ be a variable vector, and let $b \in \mathbb{R}^m$ be a known vector. Consider the system of linear equations
$$Ax = b.$$
Let E be any elementary row operation. Since E is invertible, finding a solution x to the system $Ax = b$ is equivalent to finding the solution x to the system $EAx = Eb$. By applying many elementary row operations, you have seen in a previous course how to solve the system $Ax = b$. That is, you continue to apply elementary row operations $E_1, \ldots, E_k$ such that $E_1 \cdots E_k A$ in **row-echelon** form, and you then solve $E_1 \cdots E_k Ax = E_1 \cdots E_k b$. A matrix B is in row-echelon form if each row is either zero, or its left-most nonzero entry is 1, with zeros below the 1.

Remark 3.44 (Inverting a Matrix). Let A be an invertible $n \times n$ matrix. You learned in a previous course an algorithm for inverting A using elementary row operations. Below, we will prove that this algorithm works.

Remark 3.45 (Column Operations). In the above discussion, we could have also used column operations instead of row operations. Column operations would then correspond to multiplying the matrices E on the right side, rather than the left side. The invertibility of column operations would therefore still hold.

Definition 3.46 (Rank). The **rank** of a matrix A is equal to the dimension of the space spanned by the columns of A.

Proposition 3.47. *Let A be a real $n \times n$ matrix. Then A is invertible if and only if A has rank n.*

Lemma 3.48. *Let A be a matrix in row-echelon form. Then the rank of A is equal to the number of nonzero rows of A.*

Theorem 3.49. *Let A be an $m \times n$ matrix of rank r. Then, there exist a finite number of elementary row and column operations which, when applied to A, produce the matrix*

$$\begin{pmatrix} I_{r\times r} & 0_{r\times(n-r)} \\ 0_{(m-r)\times r} & 0_{(m-r)\times(n-r)} \end{pmatrix}.$$

Proof. We first use row reduction to put A into row-echelon form. So, after this row reduction, the first r rows of A have some zeros, and then a 1 with zeros below this 1. And the remaining $m - r$ rows are all zero. (In case $r = 0$, then we are done, so we may assume that $r > 0$.) Now, the first row of A has some zeros, then a 1 with zeros below this 1. So, by adding copies of the column that contains the entry 1 to each column to the right, the remaining entries of the first row can be made to be zero. And we still keep our matrix in row-echelon form. Now, the second row of A has some zeros, then a 1 with zeros above and below this 1. So, by adding copies of the column that contains this entry 1 to each column to the right, the remaining entries of the second row can be made to be zero. And once again, our matrix is still in row-echelon form. We then continue this procedure. The first r rows then each have exactly one entry of 1, and all remaining entries in the matrix are zero. By swapping columns as needed, A is then put into the required form, as desired. $\square$

Corollary 3.50 (A Factorization Theorem). *Let A be an $m\times n$ matrix of rank r. Then, there exists an $m \times m$ matrix B and an $n \times n$ matrix C such that B is the product of a finite number of elementary row operations, C is the product of a finite number of elementary column operations, and such that*

$$A = B\begin{pmatrix} I_{r\times r} & 0_{r\times(n-r)} \\ 0_{(m-r)\times r} & 0_{(m-r)\times(n-r)} \end{pmatrix}C.$$

Lemma 3.51. *Let A be an $m \times n$ matrix. Let B be an $m \times m$ invertible matrix, and let C be an $n \times n$ invertible matrix. Then*

$$\mathrm{rank}(A) = \mathrm{rank}(BA) = \mathrm{rank}(AC) = \mathrm{rank}(BAC).$$

Lemma 3.52. *Let A be an $m \times n$ matrix with rank r. Then A^T also has rank r.*

Lemma 3.53. *Let A be an $n\times n$ matrix. Then A is invertible if and only if it is the product of elementary row and column operations.*

Remark 3.54. Suppose A is an invertible matrix, and we have elementary row operations $E_1, \ldots, E_j$ such that

$$E_1 \cdots E_j A = I_n.$$

Multiplying both sides by A^{-1} on the right,

$$E_1 \cdots E_j I_n = A^{-1}.$$

So, to compute A^{-1} from A, it suffices to find row operations that turn A into the identity. And we then apply these operations to I_n to give A^{-1}. This is the algorithm for computing the inverse A^{-1} that you learned in a previous class.

3.1.1. *Multiplying Matrices.*

Example 3.55 (Multiplying Matrices). The naïve way to multiply two real $n \times n$ matrices requires approximately n^a arithmetic operations where $a = 3$. (The output matrix has n^2 entries, and each entry requires at most $2n$ arithmetic operations, so a total of $2n \cdot n^2$ operations could be needed.) However, there seems to be some redundancy in all of these operations, so one might hope to improve the number of required operations. In fact, $a < 2.3728639$ also possible [Gal14] (building upon Coppersmith-Winograd, Stothers, and Williams.) I do not think the algorithm with such a value of a has been implemented in practice, since the implied constants in its analysis are quite large, and apparently the algorithm does not parallelize. On the other hand, Strassen's algorithm has been implemented, and it has $a = \log 7 / \log 2 \approx 2.807$.

Example 3.56 (Computing Determinants). Let $n > 0$ be an integer. Suppose we want to compute the determinant of a real $n \times n$ matrix A with entries A_{ij}, $i, j \in \{1, \ldots, n\}$. An inefficient but straightforward way to do this is to directly use a definition of the determinant. Let S_n denote the set of all permutations on n elements. For any $\sigma \in S_n$, let $\mathrm{sign}(\sigma) := (-1)^j$, where σ can be written as a composition of j transpositions (Exercise: this quantity is well-defined). (A transposition $\sigma \in S_n$ satisfies $\sigma(i) = i$ for at least $n-2$ elements of $\{1, \ldots, n\}$.) Then

$$\det(A) = \sum_{\sigma \in S_n} \mathrm{sign}(\sigma) \prod_{i=1}^{n} A_{i\sigma(i)}.$$

This sum has $|S_n| = n!$ terms. So, if we use this formula to directly compute the determinant of A, in the worst case we will need to perform at least $(n+1) \cdot n!$ arithmetic operations. This is quite inefficient. We know a better algorithm from linear algebra class. We first perform row operations on A to make it upper triangular. Suppose B is an $n \times n$ real matrix such that BA represents one single row operation on A (i.e. adding a multiple of one row to another row, or swapping the positions of two rows). Then there are real $n \times n$ matrices $B_1, \ldots, B_m$ such that

$$B_1 \cdots B_m A \qquad (*)$$

is an upper triangular matrix. The matrices $B_1, \ldots, B_m$ can be chosen to first eliminate the left-most column of A under the diagonal, then the second left-most column entries under the diagonal, and so on. That is, we can choose $m \leq n(n-1)/2$, and each row operation involves at most $3n$ arithmetic operations. So, the multiplication of $(*)$ uses at most

$$3mn \leq 2n^3$$

arithmetic operations. The determinant of the upper diagonal matrix $(*)$ is then the product of its diagonal elements, and

$$\det(B_1 \cdots B_m A) = \det(B_1) \cdots \det(B_m) \det(A).$$

That is,

$$\det(A) = \frac{\det(B_1 \cdots B_m A)}{\det(B_1) \cdots \det(B_m)}.$$

So, $\det(A)$ can be computed with at most $2n^3 + m + n \leq 4n^3 = O(n^3)$ arithmetic operations. Can we do any better?

It turns out that this is possible. Indeed, if it is possible to multiply two $n \times n$ real matrices with $O(n^a)$ arithmetic operations for some $a > 0$, then it is possible to compute the determinant of an $n \times n$ matrix with $O(n^a)$ arithmetic operations.

Remark 3.57. Interestingly, computing the permanent of a matrix

$$\mathrm{per}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i\sigma(i)}$$

is #**P**-complete, so we expect this quantity cannot be computed using a polynomial number (in n, e.g. n^3) of arithmetic operations on a computer, even though we can do this for the determinant. However, for any $\varepsilon > 0$, there is a $(1+\varepsilon)$ polynomial time randomized approximation algorithm for computing the permanent of a matrix with nonnegative entries [JSV04]. That is, for any $\varepsilon > 0$, and $0 < \delta < 1$ there is a randomized algorithm such that the following holds. For any $n \times n$ matrix A of nonnegative real numbers, the algorithm runs in time that is polynomial in $1/\varepsilon, n$, and $\log(1/\delta)$, and with probability at least $1 - \delta$ the algorithm outputs a real number p such that

$$p \leq \mathrm{per}(A) \leq (1+\varepsilon)p.$$

On the other hand, for any constant c, the problem of approximating the permanent of an arbitrary matrix A is #**P**-hard [Aar11].

Theorem 3.58 (Gaussian Elimination/ LU Factorization). *Let $\mathbb{F}$ denote $\mathbb{R}$ or $\mathbb{C}$. Let A be an $n \times n$ matrix with values in $\mathbb{F}$. Then there exist $n \times n$ matrices P, L, U such that*

$$PA = LU,$$

where L is lower triangular with ones on its diagonal and values in $\mathbb{F}$, U is upper triangular with values in $\mathbb{F}$, and P is a permutation matrix (i.e. P is the identity matrix with its rows permuted). Moreover, P, L, U can be computed with at most $5n^3$ arithmetic operations.

Remark 3.59. Even in the case $n = 1$, we can see the LU factorization is not unique.

Proof. We first apply a permutation matrix P_1 to A such that the top left entry of $P_1 A$ has largest absolute value in its first column. In the case that the first column of A is zero, let L_1 be the identity. Otherwise, let L_1 denote the (lower triangular) row operation matrix such that $L_1 P_1 A$ has zeros below the top left entry. We now iterate this procedure. Apply a permutation matrix P_2 to $L_1 P_1 A$ that fixes the first row, and such that the second diagonal entry has largest absolute value among the lowest $n-1$ entries in the second column. When all entries below and including the diagonal are zero in the second column, let L_2 be the identity. Otherwise, let L_2 denote the (lower triangular) row operation matrix (that fixes the first row) such that $L_2 P_2 L_1 P_1 A$ has zeros below the diagonal. We continue this procedure. We arrive at an upper triangular matrix U such that

$$L_{n-1} P_{n-1} \cdots L_1 P_1 A = U.$$

For each $1 \leq k \leq n-1$, define $L_k' := P_{n-1} \cdots P_{k+1} L_k P_{k+1} \cdots P_{n-1}$. Note that $L_1', \ldots, L_{n-1}'$ are lower triangular. To see this, we first write

$$P_{k+1} L_k P_{k+1} = P_{k+1}(L_k - I + I) P_{k+1} = P_{k+1}(L_k - I) P_{k+1} + I.$$

Now, $L_k - I$ is nonzero only in its k^{th} column below the k^{th} row, and P_{k+1} only permutes rows below the k^{th} row. So, $P_{k+1}(L_k - I)$ also is only nonzero in its k^{th} column below the

k^{th} row. Then multiplying on the right by P_{k+1} permutes the columns to the right of the k^{th} column (i.e. it has no effect); in block form we have

$$L_k = \begin{pmatrix} I_{k-1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & v_k & I_{n-k} \end{pmatrix}, \qquad P_{k+1} = \begin{pmatrix} I_k & 0 \\ 0 & * \end{pmatrix}.$$

Therefore,

$$\begin{aligned} P_{k+1}(L_k - I)P_{k+1} &= \begin{pmatrix} I_k & 0 \\ 0 & * \end{pmatrix} \begin{pmatrix} 0_{k-1} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & v_k & 0_{n-k} \end{pmatrix} \begin{pmatrix} I_k & 0 \\ 0 & * \end{pmatrix} \\ &= \begin{pmatrix} 0_{k-1} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & v_k' & 0_{n-k} \end{pmatrix} \begin{pmatrix} I_k & 0 \\ 0 & * \end{pmatrix} = \begin{pmatrix} 0_{k-1} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & v_k' & 0_{n-k} \end{pmatrix}. \end{aligned}$$

We conclude $P_{k+1}L_kP_{k+1}$ is lower triangular. By a similar argument, $P_{k+2}P_{k+1}L_kP_{k+1}P_{k+2}$ is lower triangular. Iterating this argument, we conclude that $L_1', \ldots, L_{n-1}'$ are all lower triangular.

By definition of $L_1', \ldots, L_{n-1}'$, we have

$$U = L_{n-1}P_{n-1} \cdots L_1P_1A = L_{n-1}' \cdots L_1'P_{n-1} \cdots P_1A.$$

Finally, define $L' := L_{n-1}' \cdots L_1'$, and define $P := P_{n-1} \cdots P_1$. We have

$$U = L'PA.$$

So, let $L := (L_1')^{-1} \cdots (L_{n-1}')^{-1}$, so that $LL' = I$, and

$$LU = LL'PA = PA.$$

(Alternatively, once we have the expression for L', we could just solve directly for its inverse, which is straightforward since L' is lower triangular, and then define $L := (L')^{-1}$ directly.)

Each iteration requires at most n^2 arithmetic operations, and there are $n-1$ such iterations, so completing the computation requires at most n^3 arithmetic operations. $\square$

Exercise 3.60. Write your own program in Python that finds the LU decomposition of a given $n \times n$ real matrix (without using any matrix decomposition programs in Python). Then apply your program to the matrix

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 2 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 1 & 0 & 2 \\ -6 & 0 & 1 & 2 & 0 \\ 3 & 0 & 4 & 0 & -1 \end{pmatrix}.$$

Exercise 3.61. Suppose A is an $n \times n$ matrix of the form

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ -1 & 1 & 0 & \cdots & 0 & 1 \\ -1 & -1 & 1 & \cdots & 0 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & -1 & -1 & \cdots & 1 & 1 \\ -1 & -1 & -1 & \cdots & -1 & 1 \end{pmatrix}$$

Find an LU decomposition of A. Hint: use

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 1 & 0 & \cdots & 0 & 0 \\ -1 & -1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & -1 & -1 & \cdots & 1 & 0 \\ -1 & -1 & -1 & \cdots & -1 & 1 \end{pmatrix}.$$

What U do you get? Explain why, when n is large, this LU decomposition would lead to a large loss of significance.

Exercise 3.62. Suppose A is an $n \times n$ real matrix with rank n. Let L_1, L_2 be real $n \times n$ lower triangular matrices, and let U_1, U_2 be real $n \times n$ upper triangular matrices. Suppose

$$A = L_1 U_1 = L_2 U_2.$$

Show that there exists a real $n \times n$ diagonal matrix D with nonzero diagonal entries such that

$$L_1 = L_2 D, \qquad U_1 = D^{-1} U_2.$$

In this sense, the LU factorization of A is unique, up to multiplication by D.

Exercise 3.63. Let L be a complex $n \times n$ lower triangular matrix with nonzero diagonal entries. Describe an algorithm that computes L^{-1} with at most $5n^3$ arithmetic operations. Then, write a program in Python that finds L^{-1} for such an $n \times n$ matrix (without using any built-in matrix inversion things in Python), and apply your program when L is

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ -6 & 0 & 1 & 2 & 0 \\ 1 & 0 & 4 & 0 & 3 \end{pmatrix}.$$

Finally, verify that your computed version of L^{-1} satisfies $LL^{-1} = L^{-1}L = I$ (at least approximately).

(Hint: L^{-1} is also lower triangular. Starting with the top row of L^{-1} and working your way down, what entries must L^{-1} have? Try starting from the diagonal entries and then moving to the left, one entry at a time.)

(Hint: it might be helpful to access portions of a row of a matrix L in Python. For example, if $1 \leq j < i \leq n$ are integers, the command `L(i, j+1: i)` is the i^{th} row of L starting from entry $j+1$ and ending at enetry i. And `L(j+1 : i, j)` is the j^{th} column of L, starting at entry $j+1$ and ending at entry i.)

Exercise 3.64 (Matrix Inversion). There are a few standard algorithms that invert an invertible matrix. One such algorithm uses the LU decomposition. Suppose A is an invertible matrix. Using the LU decomposition, show that $PA = LU$, with L, U invertible, L lower triangular, U upper triangular, and P a permutation matrix, so that $A = P^T LU$. Then, with Exercise 3.63, describe an algorithm for computing A^{-1} as $U^{-1}L^{-1}P$ that uses at most $20n^3$ arithmetic operations.

As we mentioned in Remark 3.43, the linear system $Ax = b$ can be solved (if a solution exists) by performing elementary row operations on A. These elementary row operations are encoded in the LU decomposition, so the LU decomposition can solve a linear system of equations.

Theorem 3.65 (Solving Systems of Linear Equations). *Let A be a complex $n \times n$ matrix. Let $b \in \mathbb{C}^n$. Consider the equation*

$$Ax = b$$

where $x \in \mathbb{C}^n$ is unknown. Let $PA = LU$ be the LU decomposition of A that we found (together with an algorithm) in Theorem 3.58. If a solution $x' \in \mathbb{C}^n$ exists to the equation $Ax' = b$, then some $x \in \mathbb{C}^n$ satisfying $Ax = b$ can be found by solving the following two triangular systems, whose solutions exist

- *First solve for $y \in \mathbb{C}^n$ in the equation $Ly = Pb$,*
- *Then solve for $x \in \mathbb{C}^n$ in the equation $Ux = y$, and output x,*

Remark 3.66. Solving linear triangular systems is easy. Note $LUx = PAx = Pb$.

Proof. First, note that L is lower triangular with ones on its diagonal, so a solution y exists to $Ly = Pb$ (i.e. L is invertible). Since P is invertible, $x \in \mathbb{C}^n$ satisfies $PAx = Pb$ if and only if $Ax = b$. Since $Ly = Pb$, and $PA = LU$, we have $PAx = LUx = Ly$. Since L is invertible, we have $Ux = y$. That is, $Ax = b$ is solvable for x if and only if $Ux = y$ is solvable for x. We assumed that some $x' \in \mathbb{C}^n$ solves $Ax' = b$, so we deduce this same x' solves $Ux' = y$. □

Remark 3.67. Recall that if A is a real $n \times n$ matrix with rank n, and if $b \in \mathbb{R}^n$, then the equation $Ax = b$ can always be solved for some unique $x \in \mathbb{R}^n$. More generally, if A perhaps does not have full rank, then $Ax = b$ can be solved only if b is in the span of the columns of A. In this case, the solution x might not be unique. For this reason, solving for $Ux = y$ in Theorem 3.65 might result in a non-unique solution x to $Ax = b$.

Exercise 3.68. Let

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 2 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 1 & 0 & 2 \\ -6 & 0 & 1 & 2 & 0 \\ 3 & 0 & 4 & 0 & -1 \end{pmatrix}.$$

Using the LU decomposition of A, solve the equation

$$Ax = b,$$

using Python code (without using any built-in linear algebra solvers) where $b = (2, 4, 3, 1, 5)^T$. (Note that solving a linear system such as $Ly = b$ when L is lower triangular should be relatively straightforward, by e.g. first solving for y_1, then y_2, and so on.)

Definition 3.69. A self-adjoint $n \times n$ matrix A is said to be **positive semidefinite** if all eigenvalues of A are nonnegative. If additionally all eigenvalues of A are positive, A is called **positive definite**.

Theorem 3.70. *Let A be an $n \times n$ self-adjoint matrix with values in $\mathbb{C}$. Then the following are equivalent.*

(i) *All eigenvalues of A are nonnegative.*
(ii) *There exists an $n\times n$ matrix B with values in $\mathbb{C}$ such that $A=B^*B$.*
(iii) *For any $x\in\mathbb{C}^n\setminus\{0\}$, we have $x^*Ax\geq 0$.*

Moreover, strict equality holds in (i) and (iii) if and only if all eigenvalues of A are positive.

Proof. We will show that (i) implies (ii), (ii) implies (iii), and (iii) implies (i), thereby obtaining the equivalence of all three conditions. In all cases, since A is self-adjoint, the Spectral Theorem 3.34 says $A=QDQ^*$ with D an $n\times n$ diagonal matrix with real entries and Q is unitary $n\times n$. If (i) holds, D has nonnegative entries, $A=Q\sqrt{D}\sqrt{D}Q^*=(Q\sqrt{D})(Q\sqrt{D})^*$, so (ii) holds with $B:=(Q\sqrt{D})^*$. If (ii) holds and $x\in\mathbb{C}^n\setminus\{0\}$, then $x^*Ax=x^*B^*Bx=(Bx)^*Bx=\|Bx\|^2\geq 0$, so (iii) holds. If (iii) holds and if $v\in\mathbb{C}^n\setminus\{0\}$ is an eigenvector of A with eigenvalue $\lambda\in\mathbb{R}$, then $\lambda\|v\|^2=\lambda v^*v=v^*Av\geq 0$. We conclude that $\lambda\geq 0$ since $v\neq 0$ implies $\|v\|\neq 0$ so that (i) holds. □

Corollary 3.71. *Let $\mathbb{F}$ denote $\mathbb{R}$ or $\mathbb{C}$. Let A be a self-adjoint positive definite $n\times n$ matrix with elements in $\mathbb{F}$. Then there exist $n\times n$ matrices L,U with elements in $\mathbb{F}$ such that*

$$A=LU,$$

where L is lower triangular, U is upper triangular. Moreover, L and U can be computed with at most $5n^3$ arithmetic operations.

Proof. We repeat the proof of Theorem 3.58. Observe first that the top left entry of A must be positive by Theorem 3.70(iii), so we can take $P_1=I$. Similarly, every other P_k can be taken to be the identity matrix. We prove this by contradiction. Suppose for example that at step k of the proof of Theorem 3.58, we find that all entries in the k^{th} column of the matrix below and including the k^{th} entry are all zero. If this occurs, then the top left $k\times k$ minor of $L_k\cdots L_1A$ has a zero row in its k^{th} row. So the vector $x\in\mathbb{R}^n$ with a 1 in the k^{th} entry and a zero in its other entries satisfies

$$\begin{aligned}0<x^TL_k\cdots L_1AL_1^T\cdots L_k^Tx&=x^T\begin{pmatrix}A_k&*\\0&0\end{pmatrix}L_1^T\cdots L_k^Tx\\&=x^T\begin{pmatrix}0&*\\0&0\end{pmatrix}L_1^T\cdots L_k^Tx=x^T\begin{pmatrix}0&*\\0&0\end{pmatrix}x=0.\end{aligned}$$

The first inequality used Theorem 3.70. The penultimate equality used that $L_1^T\cdots L_k^T$ is upper triangular. The last equality used the definition of x. With this contradiction, we conclude we can choose $P=I$ in Theorem 3.58.

□

Exercise 3.72. Write a computer program on your own that finds the LU factorization of the matrix

$$A=\begin{pmatrix}6&0&-4&0\\0&7&0&-1\\-4&0&6&0\\0&-1&0&7\end{pmatrix}.$$

3.2. **QR Decomposition, Eigenvalues, Power Method, QR Algorithm.**

3.2.1. *QR Decomposition.* Due to examples of LU factorizations such as Exercise 3.61, the LU decomposition (or equivalently, Gaussian elimination) might not be the best method for solving linear systems of equations. There is another matrix factorization, the QR decomposition, which sometimes behaves better for solving linear systems of equations. The QR decomposition also has other applications not directly related to solving linear systems.

We will construction the QR decomposition iteratively with the following lemma. The idea of the QR decomposition is that, rather than using row operations to force a column of a matrix to have many zeros, we will apply a (complex) rotation to the matrix to force a column to have many zeros.

Lemma 3.73. *Let $e_1 = (1, 0, \ldots, 0)^T$. Let $w \in \mathbb{C}^n$. Then there exists $v \in \mathbb{C}^n$ with $\|v\| = 1$ and there exists $\alpha \in \mathbb{C}$ such that*

$$(I - 2vv^*)w = \alpha e_1.$$

Moreover, $I - 2vv^*$ is a unitary matrix, and we can choose $\alpha := -\|w\| e^{\sqrt{-1}\theta}$ where $\theta \in [0, 2\pi)$ satisfies $w_1 = |w_1| e^{\sqrt{-1}\theta}$, i.e. θ is the angle w_1 makes with the positive real axis.

Proof. First, note that the matrix $I - 2vv^*$ is unitary since $(I - 2vv^*)v = v - 2v\|v\|^2 = -v$ and for any x perpendicular to v, we have $v^*x = 0$ so $(I - 2vv^*)x = x$, i.e. there is an orthonormal basis of $\mathbb{C}^n$ that diagonalizes $I - 2vv^*$ with all eigenvalues 1 or -1.

If $w = 0$, choose $\alpha = 0$, so below, assume $w \neq 0$. Now, we will choose α so that $vv^*w = \frac{1}{2}(w - \alpha e_1)$. Also, we will choose $v := \frac{w - \alpha e_1}{\|w - \alpha e_1\|}$ (we will choose α so the denominator is nonzero). Then

$$vv^*w = \frac{(w - \alpha e_1)(w - \alpha e_1)^*}{\|w - \alpha e_1\|^2} w = (w - \alpha e_1) \frac{\|w\|^2 - \overline{\alpha} w_1}{\|w\|^2 - \alpha \overline{w_1} - w_1 \overline{\alpha} + |\alpha|^2}.$$

We want to choose α so this quantity is $\frac{1}{2}(w - \alpha e_1)$, i.e. we choose α so that

$$2\Big(\|w\|^2 - \overline{\alpha} w_1 \Big) = \|w\|^2 - \alpha \overline{w_1} - w_1 \overline{\alpha} + |\alpha|^2. \qquad (**)$$

That is, we choose α so that

$$\|w\|^2 = -\alpha \overline{w_1} + w_1 \overline{\alpha} + |\alpha|^2 = 2\mathrm{Im}(w_1 \overline{\alpha}) + |\alpha|^2.$$

We can then choose $\alpha \in \mathbb{C}$ so that the imaginary part of $w_1 \overline{\alpha}$ is zero, and such that $|\alpha| = \|w\|$. For example, if $w_1 = re^{i\theta}$ where $i = \sqrt{-1}$, $r > 0$ and $\theta \in [0, 2\pi)$, then we can choose

$$\alpha := -\|w\| e^{i\theta},$$

so that $(**)$ holds and $\overline{\alpha} w_1 = -r\|w\|$. We now verify (recalling $w \neq 0$) that

$$\|w - \alpha e_1\|^2 = 2(\|w\|^2 - \overline{\alpha} w_1) = 2(\|w\|^2 + \|w\| r) > 0,$$

so that we have not divided by zero in the definition of α, so that $(*)$ holds as required. $\square$

Theorem 3.74 (**QR Factorization**). *Let $\mathbb{F}$ denote $\mathbb{R}$ or $\mathbb{C}$. Let A be an $m \times n$ matrix with values in $\mathbb{F}$, with $m \geq n$. Then there exists an $m \times m$ unitary matrix Q (with values in $\mathbb{F}$) and an $m \times n$ upper triangular matrix R (with values in $\mathbb{F}$) such that*

$$A = QR.$$

Moreover, Q is obtained by applying $n - 1$ unitary matrices to the columns of A, and at most $8n^2m$ arithmetic operations are required to compute the matrices Q and R.

Proof. Let w be the first column of A. We would like to apply a rotation (i.e. a unitary matrix) to A that rotates w into a vector with all entries zero except for the first entry. The unitary matrix will be $I - 2vv^*$ for some $v \in \mathbb{C}^m$ with $\|v\| = 1$. This is possible by Lemma 3.73. We then have a unitary matrix $U_1 = I - 2vv^*$ such that

$$U_1 A = \begin{pmatrix} a_1 & * \\ 0 & A_2 \end{pmatrix},$$

where A_2 is an $(m-1) \times (n-1)$ matrix. We can then iterate this procedure, finding a vector $v_2 \in \mathbb{C}^{m-1}$ such that $I - 2v_2v_2^*$ and such that $(I - 2v_2v_2^*)A_2$ has zeros below the first entry of its first column. Define then

$$U_2 := \begin{pmatrix} 1 & 0 \\ 0 & I - 2v_2v_2^* \end{pmatrix}.$$

Then U_2U_1A is upper triangular with zeros below its first two diagonal entries. After iterating this procedure $n-1$ times, we have found $m \times m$ unitary matrices $U_1, \ldots, U_{n-1}$ such that $R := U_{n-1} \cdots U_1 A$ is upper triangular. Define $Q := U_1^* \cdots U_{n-1}^*$, so that $A = QR$.

Since $(I - 2vv^*)w = w - 2v(v^*w) = w - v(2)(v^*w)$, computing $(I - 2vv^*)w$ requires at most $4m$ arithmetic operations, so that $(I - 2vv^*)A$ requires at most $4nm$ operations. Iterating $n-1$ times, we require at most $4n^2m$ operations to compute R. Since $(I - 2vv^*)^* = I - 2vv^*$, each of the unitary matrices $U_1, \ldots, U_{n-1}$ is also self-adjoint, so $Q = U_1 \cdots U_{n-1}$, and doing that multiplication also requires at most $4n^2m$ operations, for a total of at most $8n^2m$ operations. □

The above algorithm using Lemma 3.73 is preferred in practice. The QR decomposition can also be constructed via the Gram-Schmidt orthogonalization. However, the subtractions in the Gram-Schmidt procedure lead to loss of significance errors and instability.

Proof. Denote the columns of A as $A_1, \ldots, A_n$, and denote the output of Theorem 3.23 as $Q_1, \ldots, Q_n$. Let Q denote the matrix with columns $Q_1, \ldots, Q_n$. Define $r_{ij} := \langle A_j, Q_i \rangle$. Since $Q_1, \ldots, Q_n$ are an orthonormal basis of $\mathbb{C}^n$, we have by Theorem 3.20, for each $1 \leq j \leq n$,

$$A_j = \sum_{i=1}^{n} \langle A_j, Q_i \rangle Q_i = \sum_{i=1}^{n} r_{ij} Q_i.$$

That is, $a_{kj} = \sum_{i=1}^{n} q_{ki} r_{ij}$ for all $1 \leq j \leq m$, $1 \leq k \leq n$. That is, $A = QR$.

By the definition of the Gram-Schmidt procedure $r_{ii} > 0$ for all $1 \leq i \leq n$. Also, by definition of the Gram-Schmidt Orthogonalization, if $1 \leq j < i \leq n$, then Q_i is orthogonal to A_j, so that $r_{ij} = \langle Q_i, A_j \rangle = 0$. That is, R is upper triangular.

Lastly, note that the k^{th} step of the Gram-Schmidt procedure uses at most $5km$ arithmetic operations, so n total steps results in at most $n(n+1)(5/2)m$ arithmetic operations, with one final normalization step using at most $2nm$ operations, for a total of at most $5n^2m$ operations. □

Lemma 3.75. *Let A be a complex $n \times n$ matrix with rank n. Then there is a unique way to write a QR decomposition $A = QR$ where R has nonnegative diagonal entries.*

Proof. Suppose $A = QR = Q_0R_0$. Since A has rank n, R is invertible. Then $Q_0^*Q = R_0R^{-1}$. The matrix on the left is unitary and the matrix on the right is upper triangular with nonnegative diagonal entries. So, we must have $R_0R^{-1} = I$, so that $R_0 = R$ and similarly $Q = Q_0$. □

Theorem 3.70(ii) says that a self-adjoint positive definite matrix can be written as $A = B^*B$, and this factorization is useful for many applications. However, Theorem 3.70(ii) was not constructive. Below, we describe an algorithm for finding this decomposition.

Theorem 3.76 (**Cholesky Decomposition**). *Let $\mathbb{F}$ denote $\mathbb{R}$ or $\mathbb{C}$. Let A be an $n \times n$ self-adjoint positive definite matrix with values in $\mathbb{F}$. Then there exists an $n \times n$ upper triangular matrix B with elements in $\mathbb{F}$ such that*

$$A = B^*B.$$

Moreover, $B = \sqrt{(U^*)^{-1}LU}$ where $A = LU$ is an LU decomposition of A (from Corollary 3.71), so that B can be computed from Corollary 3.71 and Exercise 3.63 below. Lastly, the Cholesky decomposition $A = B^*B$ is unique up to multiplication of B by a diagonal matrix with entries with absolute value 1.

Proof. From Corollary 3.71, we can write $A = LU$ where L is lower triangular, and U is upper triangular. Since A is self-adjoint, we have

$$LU = A = A^* = U^*L^*.$$

Since A is positive definite, U, L are invertible (otherwise $0 < \det(A) = \det(L)\det(U) = 0$). So,

$$(U^*)^{-1}L = L^*U^{-1}.$$

The matrix on the left is lower triangular and the matrix on the right is upper triangular. Therefore, both of these matrices must be diagonal. That is, there is a diagonal matrix D such that $D = (U^*)^{-1}L$, i.e. $L = U^*D$. Then

$$A = LU = U^*DU. \qquad (\ddagger)$$

Then $D = (U^*)^{-1}AU^{-1} = (U^{-1})^*AU^{-1}$. From Theorem 3.70, for any $x \in \mathbb{F}^n \setminus \{0\}$,

$$x^*Dx = (U^{-1}x)^*AU^{-1}x > 0.$$

So, D is diagonal with positive values. Defining $\sqrt{D}$ as the diagonal matrix with entries the square roots of the entries of D, $(\ddagger)$ becomes

$$A = U^*\sqrt{D}\sqrt{D}U = (\sqrt{D}U)^*\sqrt{D}U.$$

We then define $B := \sqrt{D}U$. Since $B = \sqrt{(U^*)^{-1}LU}$, Corollary 3.71 and Exercise 3.63 complete the algorithm for computing B.

To see the uniqueness of the Cholesky decomposition, write $A = B^*B = C^*C$ with C, B lower triangular. Then $(CB^{-1})^*CB^{-1} = I$, so that CB^{-1} is a lower triangular, unitary matrix, i.e. $CB^{-1} = D$ where D is a diagonal matrix with entries with absolute value 1, so that $C = DB$. □

3.2.2. *Matrix Norms as a Measure of Error.* Norms are used in numerical analysis to bound the errors in matrix computations.

An $n \times m$ matrix can be treated as a vector of length nm, so that a set of matrices can be equipped with a vector norm.

However, there are also other natural norms on the set of matrices, e.g. ones related to eigenvalues or singular values of the matrix, which we describe further below.

Definition 3.77 (Singular Values). Let A be an $m \times n$ complex matrix. Then the **singular values** of A are the square roots of the eigenvalues of A^*A. (Theorem 3.70 says the eigenvalues of A^*A are nonnegative.)

Definition 3.78 (Vector ℓ_p Norms). Let $1 \leq p < \infty$. Let $x \in \mathbb{C}^n$. Define the ℓ_p norm of x to be

$$\|x\|_p := \Big(\sum_{i=1}^n |x_i|^p\Big)^{1/p}.$$

Also define the ℓ_∞ norm of x to be

$$\|x\|_\infty := \max_{1\leq i\leq n} |x_i|\,.$$

Proposition 3.79. *The ℓ_p norm is a norm for any $1 \leq p \leq \infty$, i.e. the definition of a norm from Definition 3.8 holds.*

Proof. We will show the triangle inequality holds. The case $p = \infty$ follows from the scalar triangle inequality, so assume $1 \leq p < \infty$. Let $x, y \in \mathbb{C}^n$. We need to show that $\|x+y\|_p \leq \|x\|_p + \|y\|_p$. By scaling, we may assume $\|x\|_p = 1-t$, $\|y\|_p = t$, for some $t \in (0,1)$ (zeros and infinities being trivial). Define $v := x/(1-t)$, $w := y/t$. Then by convexity of $s \mapsto |s|^p$ on $\mathbb{R}$,

$$|(1-t)v_i + tw_i|^p \leq (1-t)\,|v_i|^p + t\,|w_i|^p\,, \qquad \forall\, 1 \leq i \leq n$$

Summing over $1 \leq i \leq n$, we get

$$\|x+y\|_p^p \leq (1-t)\,\|v\|_p^p + t\,\|w\|_p^p = (1-t)+t = 1.$$

So, $\|x+y\|_p \leq 1 = \|x\|_p + \|y\|_p$. □

Definition 3.80 (Standard Inner Product). Let $x, y \in \mathbb{C}^n$. Define the standard inner product of x and y by

$$\langle x, y\rangle := \sum_{i=1}^n x_i\overline{y_i}.$$

One can check that the standard inner product is an inner product, i.e. Definition 3.10 holds in this case.

Theorem 3.81 (Hölder's Inequality). *Let $1 \leq p \leq \infty$, and let q be dual to p (so $1/p + 1/q = 1$). Let $x, y \in \mathbb{C}^n$. Then*

$$|\langle x, y\rangle| \leq \|x\|_p\,\|y\|_q\,.$$

The case $p = q = 2$ recovers the ***Cauchy-Schwarz*** *inequality:*

$$|\langle x, y\rangle| \leq \|x\|_2\,\|y\|_2\,.$$

Proof. By scaling, we may assume $\|x\|_p = \|y\|_q = 1$ (zeros and infinities being trivial). Also, the case $p = 1, q = \infty$ follows from the triangle inequality, so we assume $1 < p < \infty$. From concavity of the log function, we have

$$|x_i y_i| = (|x_i|^p)^{1/p}(|y_i|^q)^{1/q} \leq \frac{1}{p}|x_i|^p + \frac{1}{q}|y_i|^q, \qquad \forall\, 1 \leq i \leq n.$$

Summing over $1 \leq i \leq n$, we get $|\langle x, y\rangle| \leq \frac{1}{p} + \frac{1}{q} = 1 = \|x\|_p \|y\|_q$. □

Corollary 3.82 (Duality for ℓ_p Norms). *Let $1 \leq p \leq \infty$, and let q be dual to p (so $1/p + 1/q = 1$). Let $y \in \mathbb{C}^n$. Then*

$$\|y\|_p = \sup_{x \in \mathbb{R}^n \colon \|x\|_q \leq 1} |\langle x, y\rangle|.$$

Proof. Hölder's inequality, Theorem 3.81, implies that $\sup_{x \in \mathbb{R}^n \colon \|x\|_q \leq 1} |\langle x, y\rangle| \leq \|y\|_p$. To get the other corresponding inequality, consider $x \in \mathbb{C}^n$ defined by $x_i := \|y\|_p^{-(p-1)}\, \overline{y_i}\, |y_i|^{p-2}\, 1_{y_i \neq 0}$ for all $1 \leq i \leq n$. Since $1/p + 1/q = 1$, $p + q = pq$, and $q(p-1) = p$, so

$$\|x\|_q^q = \|y\|_p^{-q(p-1)} \sum_{i=1}^n |y_i|^{q(p-1)} = \|y\|_p^{-p} \sum_{i=1}^n |y_i|^p = \|y\|_p^{p-p} = 1.$$

$$\langle x, y\rangle = \|y\|_p^{-(p-1)} \sum_{i=1}^n |y_i|^p = \|y\|_p.$$

Therefore, $\sup_{x \in \mathbb{R}^n \colon \|x\|_q \leq 1} |\langle x, y\rangle| \geq \|y\|_p$. □

A natural class of matrix norms is defined in analogy with Corollary 3.82.

Definition 3.83. Let A be an $m \times n$ complex matrix. Let $1 \leq p, q \leq \infty$. Define the $p \to q$ norm of A to be

$$\|A\|_{p \to q} := \sup_{x \in \mathbb{C}^n \colon \|x\|_p \leq 1} \|Ax\|_q.$$

Proposition 3.84. *The $p \to q$ norm is a norm for any $1 \leq p, q \leq \infty$, i.e. the definition of a norm from Definition 3.8 holds.*

Proof. We will show the triangle inequality holds. Let A, B be $m \times n$ complex matrices. Fix $x \in \mathbb{C}^n$ with $\|x\|_p \leq 1$. From the triangle inequality for the ℓ_q norm, $\|(A + B)x\|_q \leq \|Ax\|_q + \|Bx\|_q$. Taking the supremum over x on both sides, we get $\|A + B\|_{p \to q} \leq \|A\|_{p \to q} + \|B\|_{p \to q}$. □

The supremum definition makes these norms difficult to compute directly. Still, in certain cases, Corollary 3.82 can give simpler expressions for these norms.

Exercise 3.85. Let A be an $m \times n$ complex matrix. Show the following

- $\|A\|_{1 \to 1} = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$.
- $\|A\|_{\infty \to \infty} = \max_{1 \leq i \leq n} \sum_{j=1}^m |a_{ij}|$.
- $\|A\|_{2 \to 2}^2$ is equal to the largest eigenvalue of AA^* (or of A^*A). That is, $\|A\|_{2 \to 2}$ is the largest singular value of A.
- For any $1 \leq p, q \leq \infty$, $\|AB\|_{p \to q} \leq \|A\|_{p \to q} \|B\|_{p \to q}$.

Theorem 3.86 ([GVL13, Theorem 3.3.1]). *Let $\|\cdot\|$ denote any matrix norm. Let A be an $n \times n$ complex matrix. Assume that A has an LU factorization of the form $A = LU$, and the diagonal entries of L and U are all positive. Assume*

- *All operations use normal floating point numbers.*
- *For any floating point numbers x, y used in the algorithm, for any operation $\odot$ among addition, subtraction, multiplication, and division, we have*
$$\mathrm{fl}(x \odot y) = x \odot_{\mathrm{fl}} y.$$
Here $\odot_{\mathrm{fl}}$ denotes the floating point implementation of an operation such as addition.
- *For any floating point numbers x, y used in the algorithm, for any operation $\odot$ among addition, subtraction, multiplication, and division, there exists $\delta \in \mathbb{R}$ with $|\delta| \leq \varepsilon$ such that*
$$x \odot_{\mathrm{fl}} y = (x \odot y)(1 + \delta).$$

Then the algorithm in Theorem 3.58 outputs a factorization $\widetilde{L}, \widetilde{U}$ such that
$$\|A - \widetilde{L}\widetilde{U}\| \leq 3(1 - (1 + 2^{-52})^n)\Big(\|A\| + \|L\| \, \|U\| \Big).$$

Recall from Exercise 3.61 that $\|L\|$ or $\|U\|$ can be exponentially large in n, in which case this theorem has limited significance.

3.2.3. *Power Method.*

Exercise 3.87 (The Power Method). This exercise gives an algorithm for finding the eigenvectors and eigenvalues of a symmetric matrix. In modern statistics, this is often a useful thing to do. The Power Method described below is not the best algorithm for this task, but it is perhaps the easiest to describe and analyze.

Let A be an $n \times n$ real symmetric matrix. Let $\lambda_1 \geq \cdots \geq \lambda_n$ be the (unknown) eigenvalues of A, and let $v_1, \ldots, v_n \in \mathbb{R}^n$ be the corresponding (unknown) eigenvectors of A such that $\|v_i\| = 1$ and such that $Av_i = \lambda_i v_i$ for all $1 \leq i \leq n$.

Given A, our first goal is to find v_1 and λ_1. For simplicity, assume that $1/2 < \lambda_1 < 1$, and $0 \leq \lambda_n \leq \cdots \leq \lambda_2 < 1/4$. Suppose we have found a vector $v \in \mathbb{R}^n$ such that $\|v\| = 1$ and $|\langle v, v_1\rangle| > 1/n$. (An exercise more suitable for a probability class shows that a randomly chosen v satisfies this property, with probability at least $1/2$.) Let k be a positive integer. Show that
$$A^k v$$
approximates v_1 well as k becomes large. More specifically, show that for all $k \geq 1$,
$$\left\|A^k v - \langle v, v_1\rangle \lambda_1^k v_1\right\|^2 \leq \frac{n-1}{16^k}.$$
(Hint: use the spectral theorem for symmetric matrices.)

Since $|\langle v, v_1\rangle| \, \lambda_1^k > 2^{-k}/n$, this inequality implies that $A^k v$ is approximately an eigenvector of A with eigenvalue λ_1. That is, by the triangle inequality,
$$\left\|A(A^k v) - \lambda_1(A^k v)\right\| \leq \left\|A^{k+1} v - \langle v, v_1\rangle \lambda_1^{k+1} v_1\right\| + \lambda_1 \left\|\langle v, v_1\rangle \lambda_1^k v_1 - A^k v\right\| \leq 2\frac{\sqrt{n-1}}{4^k}.$$
Moreover, by the reverse triangle inequality,
$$\left\|A^k v\right\| = \left\|A^k v - \langle v, v_1\rangle \lambda_1^k v_1 + \langle v, v_1\rangle \lambda_1^k v_1\right\| \geq \frac{1}{n} 2^{-k} - \frac{\sqrt{n-1}}{4^k}.$$

In conclusion, if we take k to be large (say $k > 10\log n$), and if we define $z := A^k v$, then z is approximately an eigenvector of A, that is

$$\left\| A\frac{A^k v}{\|A^k v\|} - \lambda_1 \frac{A^k v}{\|A^k v\|}\right\| \leq 4n^{3/2}2^{-k} \leq 4n^{-4}.$$

And to approximately find the first eigenvalue λ_1, we simply compute

$$\frac{z^T A z}{z^T z}.$$

That is, we have approximately found the first eigenvector and eigenvalue of A.

Remarks. To find the second eigenvector and eigenvalue, we can repeat the above procedure, where we start by choosing v such that $\langle v, v_1\rangle = 0$, $\|v\| = 1$ and $|\langle v, v_2\rangle| > 1/(10\sqrt{n})$. To find the third eigenvector and eigenvalue, we can repeat the above procedure, where we start by choosing v such that $\langle v, v_1\rangle = \langle v, v_2\rangle = 0$, $\|v\| = 1$ and $|\langle v, v_3\rangle| > 1/(10\sqrt{n})$. And so on.

Google's PageRank algorithm uses the power method to rank websites very rapidly. In particular, they let n be the number of websites on the internet (so that n is roughly 10^9). They then define an $n \times n$ matrix C where $C_{ij} = 1$ if there is a hyperlink between websites i and j, and $C_{ij} = 0$ otherwise. Then, they let B be an $n \times n$ matrix such that B_{ij} is 1 divided by the number of 1's in the i^{th} row of C, if $C_{ij} = 1$, and $B_{ij} = 0$ otherwise. Finally, they define

$$A = (.85)B + (.15)D/n$$

where D is an $n \times n$ matrix all of whose entries are 1.

The power method finds the eigenvector v_1 of A, and the size of the i^{th} entry of v_1 is proportional to the "rank" of website i.

Exercise 3.88. Consider the following symmetric real matrix

$$A = \begin{pmatrix} 5 & 1 & -2 & 3 & 1 \\ 1 & 3 & 6 & 0 & 0 \\ -2 & 6 & 0 & 1 & 1 \\ 3 & 0 & 1 & 1 & 2 \\ 1 & 0 & 1 & 2 & 3 \end{pmatrix}.$$

Using the power method (i.e. by examining large powers of A in Python), find the largest eigenvalue $\lambda \in \mathbb{R}$ of A and a corresponding eigenvector $v \in \mathbb{R}^5$ with $\|v\|_2 = 1$.

Note that $(A - \lambda vv^T)v = Av - \lambda v = 0$, and if w is any other eigenvector of A, then $(A - \lambda vv^T)w = Aw$. Using this observation, apply the power method to $A - \lambda vv^T$ to find the second largest eigenvalue of A.

Finally, compare your results with the built-in Python function `np.linalg.eig`.

3.2.4. *Eigenvalues and the QR Algorithm.* The power method just described can find the first few eigenvalues and eigenvectors of a self-adjoint matrix relatively quickly. However, finding all eigenvalues and eigenvectors with this method can be costly. Thankfully, there is an efficient way to find all eigenvalues and eigenvectors of a matrix simultaneously, using a cleverly chosen sequence of QR decompositions.

Algorithm 3.89 (**QR Algorithm for Eigenvalues**). **Input**: A symmetric $n \times n$ real matrix A (or a self-adjoint complex matrix A), a number of iterations k.
Output: An $n \times n$ matrix D' whose diagonal entries approximate the eigenvalues of A.
Define $A_0 := A$. For each $1 \leq j \leq k$, do the following.

- Write A_{j-1} in its QR Factorization as $A_{j-1} =: Q_j R_j$ (using the algorithm from Theorem 3.74, which uses either Householder reflections (i.e. Lemma 3.73) or Gram-Schmidt orthogonalization).
- Define $A_j := R_j Q_j$.

Output $D' := A_k$.

Intuition. When $A_{j-1} = Q_j R_j$, if (hypothetically) $R_j = DQ_j^T$, then $R_j Q_j = DQ_j^T Q_j = D$. That is, reversing the order in the QR factorization might produce something close to a diagonal matrix.

Theorem 3.90. *Let A be a real symmetric $n \times n$ positive definite matrix with distinct eigenvalues $\lambda_1 > \cdots > \lambda_n > 0$. (From the spectral theorem 3.34, write $A = QDQ^{-1}$ where Q is an orthogonal matrix.) Assume that D is ordered so that $D_{ii} = \lambda_i$ for all $1 \leq i \leq n$. Assume that Q^T has an LU decomposition $Q^T = LU$ where the diagonal entries of U are positive.*

Then as $k \to \infty$, the sequence of matrices $A_1, A_2, \ldots$ in Algorithm 3.89 converges to D, and $Q_1 \cdots Q_k$ converges to Q.

Proof. Note that $A = A_0 = Q_1 R_1$ and

$$A^2 = Q_1 R_1 Q_1 R_1 = Q_1 A_1 R_1 = Q_1 Q_2 R_2 R_1.$$

More generally, we can prove by induction on k that

$$A^k = Q_1 \cdots Q_k R_k \cdots R_1. \qquad (\ddagger)$$

Since $A = QDQ^{-1}$, $A^k = QD^kQ^{-1}$, so recalling $Q^T = LU$, so $L = Q^T U^{-1}$,

$$QD^kLD^{-k} = QD^kQ^TU^{-1}D^{-k} = A^kU^{-1}D^{-k} \overset{(\ddagger)}{=} Q_1 \cdots Q_k R_k \cdots R_1 U^{-1} D^{-k}. \qquad (**)$$

Recall that the diagonal entries of L are all 1. For any $1 \leq i, j \leq n$, note that

$$(D^kLD^{-k})_{ij} = \begin{cases} 1 & \text{, if } i = j \\ L_{ij}\left(\frac{\lambda_i}{\lambda_j}\right)^k & \text{, if } i > j \\ 0 & \text{, otherwise.} \end{cases}$$

Since $\lambda_i < \lambda_j$ when $i > j$, D^kLD^{-k} converges to the identity matrix as $k \to \infty$. So, QD^kLD^{-k} converges to Q as $k \to \infty$, and $(**)$ implies that $Q_1 \cdots Q_k R_k \cdots R_1 U^{-1} D^{-k}$ converges to Q as $k \to \infty$. The matrix Q itself has a unique QR factorization as $Q = Q \cdot I$ with nonnegative diagonal entries on the second term (by Lemma 3.75). So, as $k \to \infty$, $Q_1 \cdots Q_k$ converges to Q (the diagonal entries of D and D^{-1} are positive). Multiplying both sides by Q_k^{-1}, $Q_1 \cdots Q_{k-1}$ converges to QQ_k^{-1} as $k \to \infty$ as well. So, as $k \to \infty$, Q_k converges to I. From Algorithm 3.89, $A_k = Q_k R_k$. It also follows by induction that A_k is symmetric and similar to $A = A_0$ (we know $A = A_0$ is symmetric, and Algorithm 3.89 says $A_k = R_k Q_k = Q_k^T Q_k R_k Q_k = Q_k^T A_{k-1} Q_k$.) Since $A_k = Q_k R_k$, A_k is symmetric, and Q_k converges to I as $k \to \infty$, it follows that A_k converges to a diagonal matrix as $k \to \infty$.

Since A_k is similar to A, as $k \to \infty$, A_k converges to a diagonal matrix whose elements are the eigenvalues of A. Since $A_k = (Q_1 \cdots Q_k)^T A (Q_1 \cdots Q_k)$ and $Q_1 \cdots Q_k$ converges to Q as $k \to \infty$, we must have A_k converging to D as $k \to \infty$, since $A = QDQ^T$, i.e. $Q^T AQ = D$.

□

Remark 3.91. This argument shows that A_k converges exponentially fast to a diagonal matrix whose elements are the eigenvalues of A, in theory.

Exercise 3.92. Consider the following symmetric real matrix

$$A = \begin{pmatrix} 5 & 1 & -2 & 3 & 1 \\ 1 & 3 & 6 & 0 & 0 \\ -2 & 6 & 0 & 1 & 1 \\ 3 & 0 & 1 & 1 & 2 \\ 1 & 0 & 1 & 2 & 3 \end{pmatrix}.$$

Using the QR algorithm, find all eigenvalues and eigenvectors of A.

Finally, compare your results with the built-in Python function `np.linalg.eig`.

In order to decrease the computation time in the QR algorithm, the matrix A can be pre-processed into a similar tridiagonal matrix. This step can be done by a modification of the QR factorization. If A is a symmetric real $n \times n$ matrix, and $w \in \mathbb{R}^{n-1}$ is the lowest $n-1$ entries of the first column of A, then Lemma 3.73 says we can find $v \in \mathbb{C}^{n-1}$ such that the $(n-1) \times (n-1)$ unitary matrix $I - 2vv^*$ satisfies $(I - 2vv^*)w = \alpha e_1$. So, if

$$Q := \begin{pmatrix} 1 & 0 \\ 0 & I - 2vv^* \end{pmatrix},$$

then QA has a first column with zeros below its first two entries. But then QAQ^T also has a first column with zeros below its first two entries, since multiplying on the right by Q^T has no effect on those zero entries. Since A is symmetric, QAQ^T then must have zeros in its first row after its first two entries. In summary, QAQ^T has the form

$$QAQ^T = \begin{pmatrix} * & * & 0 & \cdots & 0 \\ * & * & * & \cdots & * \\ 0 & * & * & \cdots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & \cdots & * \end{pmatrix}$$

We can then iterate this procedure, letting w be the lowest $n-2$ entries of the second column of QAQ^T. After $n-1$ iterations, we obtain a tridiagonal matrix QAQ^T. The NCM package command `eigsvdgui` illustrates this procedure.

3.3. **Least Squares.** In Theorem 3.65, we used the LU decomposition of a square matrix A to solve the equation $Ax = b$, if a solution x exists. If A does not have full rank, then a solution to the equation $Ax = b$ might not exist. In such a case, we still might want to find a vector x that "most closely" solves the equation $Ax = b$. More precisely, we want to find a vector x that minimizes the quantity $\|Ax - b\|_2$, or equivalently, $\|Ax - b\|_2^2$.

Definition 3.93 (**Least Squares Problem**)**.** Let A be an $m \times n$ real matrix. Let $b \in \mathbb{R}^m$. The **least squares** problem asks for a vector $x \in \mathbb{R}^n$ minimizing

$$\|Ax - b\|_2^2 .$$

Example 3.94. Suppose we have data points $(a_1, b_1), \ldots, (a_m, b_m) \in \mathbb{R}^2$ and we would like to find the "best fit" line to the data. More specifically, we would like to find $x_0, x_1 \in \mathbb{R}$ such that the linear function

$$f(t) := x_0 + x_1 t, \qquad \forall\, t \in \mathbb{R}$$

minimizes the sum of squared differences

$$\sum_{i=1}^{m} [f(a_i) - b_i]^2 = \sum_{i=1}^{m} [x_0 + x_1 a_i - b_i]^2.$$

We can rewrite this sum of squares in matrix form as

$$\|Ax - b\|^2$$

where $x = \binom{x_0}{x_1}$, $b = (b_1, \ldots, b_m)^T$, and

$$A = \begin{pmatrix} 1 & a_1 \\ 1 & a_2 \\ \vdots & \vdots \\ 1 & a_m \end{pmatrix}.$$

So, finding the "best fit" line is a special case of the least squares minimization problem.

More generally, let $k \geq 1$ be an integer, and suppose we would like to we would like to find the "best fit" degree k polynomial to the data, i.e. we would like to find $x_0, \ldots, x_k \in \mathbb{R}$ such that the polynomial

$$f(t) := x_0 + x_1 t + \cdots + x_k t^k, \qquad \forall\, t \in \mathbb{R}$$

minimizes the sum of squared differences

$$\sum_{i=1}^{m} [f(a_i) - b_i]^2 = \sum_{i=1}^{m} [x_0 + x_1 a_i + \cdots + x_k a_i^k - b_i]^2.$$

We can rewrite this sum of squares in matrix form as

$$\|Ax - b\|^2$$

where $x = (x_0, \ldots, x_k)^T$, $b = (b_1, \ldots, b_m)^T$, and A is a Vandermonde matrix

$$A = \begin{pmatrix} 1 & a_1 & a_1^2 & \cdots & a_1^k \\ 1 & a_2 & a_2^2 & \cdots & a_2^k \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_m & a_m^2 & \cdots & a_m^k \end{pmatrix}.$$

As before, finding the "best fit" degree k polynomial is a special case of the least squares minimization problem.

Lemma 3.95. *Let A be an $m \times n$ real matrix. Let $b \in \mathbb{R}^m$.*

The vector $x \in \mathbb{R}^n$ minimizes $\|Ax - b\|$ if and only if $A^T A x = A^T b$.

Proof. Let $t \in \mathbb{R}$ and consider the function $f \colon \mathbb{R} \to \mathbb{R}$ defined by

$$f(t) := \|A(x + ty) - b\|_2^2 = \|Ax - b + tAy\|_2^2 = \|Ax - b\|_2^2 + 2ty^T A^T (Ax - b) + t^2 \|Ay\|_2^2.$$

This function is quadratic in t, and a minimum occurs at $t = 0$ if and only if $y^T A^T (Ax - b) = 0$ for all $y \in \mathbb{R}^n$, i.e. when $A^T(Ax - b) = 0$. $\square$

Exercise 3.96. Let A be an $m \times n$ real matrix with $m \geq n$. Show that A has rank n if and only if $A^T A$ is positive definite.

(Hint: $A^T A$ is always positive semidefinite.)

Lemma 3.95 and Exercise 3.96 together imply the following.

Proposition 3.97. *Let A be an $m \times n$ real matrix with $m \geq n$. Suppose A has rank n. Then there is a unique solution of the least squares problem given by*

$$x = (A^T A)^{-1} A^T b.$$

In this case, the matrix $(A^T A)^{-1} A^T$ is called the ***pseudoinverse*** *of A. (When A is a non-square matrix, it will not have an inverse, but $(A^T A)^{-1} A^T A = I$.)*

Explicitly inverting a matrix is generally not advisable. For this reason, when m, n are large, it is not a good idea to use Proposition 3.97 and its explicit formula for the x minimizing the least squares problem. We can instead minimize a full rank least squares problem using the two methods described below.

Theorem 3.98 (Least Squares via Cholesky Decomposition). *Let A be an $m \times n$ real matrix with $m \geq n$. Suppose A has rank n. The unique solution of the least squares problem can be found in the following way*

- *Find the Cholesky decomposition $L^* L$ of $A^* A$, where L is an $n \times n$ lower triangular real matrix (using the algorithm from Theorem 3.76, which uses an LU factorization of $A^* A$.)*
- *Solve the equation $L^* y = A^* b$ for $y \in \mathbb{R}^n$.*
- *Solve the equation $Lx = y$ for $x \in \mathbb{R}^n$.*

Proof. From Lemma 3.95 and Proposition 3.97, there is a unique solution $x \in \mathbb{R}^n$ to the equation $A^* A x = A^* b$. By assumption, we have $L^* L x = A^* b$. Since $A^* A$ is positive definite by Exercise 3.96, L is invertible, so there is a unique solution y to the equation $L^* y = A^* b$. Similarly, there is a unique solution x' to $Lx' = y$. Once these equations are solved, we have $A^* b = L^* y = L^* L x'$. That is, $x = x'$. $\square$

Theorem 3.99 (Least Squares via QR Decomposition). *Let A be an $m \times n$ real matrix with $m \geq n$. Suppose A has rank n. The unique solution of the least squares problem can be found in the following way*

- *Find the QR decomposition QR of A, (using the algorithm from Theorem 3.74, which uses either Householder reflections (i.e. Lemma 3.73) or Gram-Schmidt orthogonalization.)*
- *Solve the equation $Rx = Q^* b$ for $x \in \mathbb{R}^n$.*

Proof. From Lemma 3.95 and Proposition 3.97, there is a unique solution $x \in \mathbb{R}^n$ to the equation $A^* A x = A^* b$. Since $A^* A = R^* Q^* Q R = R^* R$, we can rewrite the first equation as $R^* R x = R^* Q^* b$. Since R is $n \times n$ and upper triangular, and A has rank n, R is invertible, so we can rewrite the equation as $Rx = Q^* b$. $\square$

Exercise 3.100. Suppose we have data points $(0,1), (1,3), (2,3), (3,5), (4,2) \in \mathbb{R}^2$ denoted as $\{(a_i, b_i)\}_{i=1}^5$. Find the line that best fits the data. That is, find the line $f \colon \mathbb{R} \to \mathbb{R}$ that minimizes the sum of squared differences $\sum_{j=1}^5 |f(a_i) - b_i|^2$. Use either a Cholesky decomposition or a QR decomposition, with your own method written in Python (i.e. don't use any built-in Python matrix decomposition functions).

Then, find the best fit degree two polynomial, and the best fit degree three polynomial to these data points.

3.4. **Singular Value Decomposition (SVD).**

Recall the definition of singular values in Definition 3.77.

Remark 3.101. If $m = n$, if A is self-adjoint, and if $\lambda \in \mathbb{R}$ is an eigenvalue of A with eigenvector $v \in \mathbb{C}^n$, then $A^*Av = A^2v = \lambda^2 v$, so that $|\lambda|$ is a singular value of A.

The Spectral Theorem 3.33 3.34 says that a square matrix A can be written as $A = QDQ^{-1}$ under some assumptions. In general, not every matrix can be written in this way. However, a different decomposition, known as the singular value decomposition, can be applied to any matrix.

Theorem 3.102 (**Singular Value Decomposition (SVD)**). *Let $\mathbb{F}$ denote $\mathbb{R}$ or $\mathbb{C}$. Let A be an $m \times n$ matrix with values in $\mathbb{F}$ with $m \leq n$. Then there exists a $p \times p$ diagonal matrix D with positive entries ($p \leq m$), an $m \times m$ unitary matrix U, an $n \times n$ unitary matrix V, each with elements in $\mathbb{F}$ such that*

$$A = U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} V.$$

Moreover, $AA^* = U \begin{pmatrix} D^2 & 0 \\ 0 & 0 \end{pmatrix} U^*$ and $V = \begin{pmatrix} (D^{-1},0)U^*A \\ \cdots \end{pmatrix}$ where D has no zero diagonal entries. (And U, D can be obtained from the QR Algorithm 3.89 applied to AA^*.) (In the case $p = m$ we have $A = U(D, 0)V$, $AA^* = UD^2U^*$, etc.)

Proof. Theorem 3.70 implies that AA^* and A^*A are self-adjoint positive semidefinite. The Spectral Theorem 3.34 implies that unitary $m \times m$ U and diagonal $p \times p$ D with positive entries exists such that $p \leq m$ and

$$AA^* = U \begin{pmatrix} D^2 & 0 \\ 0 & 0 \end{pmatrix} U^*. \qquad (\ddagger)$$

We can apply the QR Algorithm 3.89 and Theorem 3.90 to AA^* to compute U, D in the factorization $AA^* = U \begin{pmatrix} D^2 & 0 \\ 0 & 0 \end{pmatrix} U^*$ (at least when all entries of D are positive and distinct).

Recall D is diagonal with positive entries so D^{-1} exists. Define $Z := (D^{-1}, 0)U^*A$. Then

$$ZZ^* = (D^{-1}, 0)U^*AA^*U \begin{pmatrix} D^{-1} \\ 0 \end{pmatrix} = (D^{-1}, 0) \begin{pmatrix} D^2 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} D^{-1} \\ 0 \end{pmatrix} = D^{-1}D^2D^{-1} = I.$$

By its definition, Z is an $m \times n$ matrix with m orthogonal rows. Since $m \leq n$, we can add extra rows to Z as necessary to obtain an $n \times n$ matrix V with orthogonal rows.

Finally, observe that

$$\begin{aligned} U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} V &= U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} Z \\ \cdots \end{pmatrix} = U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} (D^{-1}, 0)U^*A \\ \cdots \end{pmatrix} = U \begin{pmatrix} D \\ 0 \end{pmatrix} (D^{-1}, 0)U^*A \\ &= U \begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix} U^*A = U\Big(I - \begin{pmatrix} 0 & 0 \\ 0 & I \end{pmatrix}\Big)U^*A = A - U \begin{pmatrix} 0 & 0 \\ 0 & I \end{pmatrix} U^*A = A. \end{aligned}$$

In the last line we used $U\begin{pmatrix}0 & 0\\ 0 & I\end{pmatrix}U^*A = 0$, which follows since

$$U\begin{pmatrix}0 & 0\\ 0 & I\end{pmatrix}U^*AA^* \overset{(\ddagger)}{=} U\begin{pmatrix}0 & 0\\ 0 & I\end{pmatrix}\begin{pmatrix}D^2 & 0\\ 0 & 0\end{pmatrix}U^* = U0U^* = 0. \qquad (**)$$

Therefore,

$$U\begin{pmatrix}0 & 0\\ 0 & I\end{pmatrix}U^*A\Big[U\begin{pmatrix}0 & 0\\ 0 & I\end{pmatrix}U^*A\Big]^* = \Big[U\begin{pmatrix}0 & 0\\ 0 & I\end{pmatrix}U^*AA^*\Big](\cdots) \overset{(**)}{=} 0,$$

so that $U\begin{pmatrix}0 & 0\\ 0 & I\end{pmatrix}U^*A = 0$. We have therefore shown the existence of the SVD. Lastly, in order to conclude that V is a unitary matrix, we have used Lemma 3.103 below. □

Lemma 3.103. *Let C be an $n \times n$ complex matrix such that $CC^* = I$. Then $C^*C = I$.*

Proof. By assumption, we have $C^*CC^*C = C^*C$, i.e. $(C^*C)^2 = C^*C$. Since C^*C is a self-adjoint positive semidefinite matrix, the spectral theorem 3.34 implies that there is a unitary $n \times n$ matrix U and a diagonal matrix D with nonnegative diagonal entries such that $C^*C = UDU^*$. Since $(C^*C)^2 = C^*C$, we have

$$UD^2U^* = UDU^*.$$

That is, $D^2 = D$. Since D is diagonal with nonnegative diagonal entries, we conclude that $D = I$, so that $C^*C = UIU^* = UU^* = I$. □

Exercise 3.104.

- Give an example of a real 2×2 matrix A with a non-unique singular value decomposition. That is, find a diagonal 2×2 matrix D, unitary 2×2 matrices $U_1 \neq U_2$, unitary 2×2 matrices $V_1 \neq V_2$ such that
$$A = U_1DV_1 = U_2DV_2.$$
- Give an example of a real 2×3 matrix A with a non-unique singular value decomposition. That is, find a diagonal 2×2 matrix D, unitary 2×2 matrices $U_1 \neq U_2$, unitary 3×3 matrices $V_1 \neq V_2$ such that
$$A = U_1(D,\, 0)V_1 = U_2(D,\, 0)V_2.$$

3.4.1. *Additional Comments.* We noted in Theorem 3.86 a performance guarantee of the LU factorization. Yet this guarantee can be quite bad in the worst case, due to the example from Exercise 3.61. On the other hand, there are various known average-case guarantees for the LU factorization, such as: https://arxiv.org/abs/2206.01726

We remarked above that the matrix $p \to q$ norms can be difficult to compute exactly for certain p, q. For more precise computational hardness statements, see e.g.
https://arxiv.org/abs/1802.07425 and https://epubs.siam.org/doi/10.1137/S0097539704441629.

For very large matrices, SVD decompositions are difficult to compute efficiently, due to the need to multiply large matrices to compute the SVD. To alleviate this issue, we can try to reduce the dimension of the matrices while preserving their structure. For more on this topic see e.g. https://arxiv.org/abs/0909.4061 or https://en.wikipedia.org/wiki/Johnson Lindenstrauss lemma

The LU and QR decomposition solve linear systems of equations relatively quickly, though one could try to decrease their computation times even more. For more on this topic, see e.g. https://arxiv.org/abs/2007.10254.

4. Clustering

4.1. **Principal Component Analysis (PCA).**

Principal Component Analysis is a procedure for reducing the dimension of data points, without losing too much "information" in the data. As we will see, PCA is an application of the singular value decomposition.

Algorithm 4.1 (**Principal Component Analysis**)**.** Input: positive integers $m \leq n$, vectors $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^n$, an integer $1 \leq q < n$. Output: vectors $y^{(1)}, \ldots, y^{(m)} \in \mathbb{R}^q$

- Let A be the $m \times n$ matrix whose rows are $x^{(1)}, \ldots, x^{(m)}$.
- From Theorem 3.102, write a singular value decomposition of A as
$$A = U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} V,$$
where U is $m \times m$, D is $p \times p$ ($p \leq m$), and V is $n \times n$. Assume that $D_{ii} \geq D_{i+1,i+1}$ for all $1 \leq i \leq p-1$. (Recall U, V are unitary by Theorem 3.102.)
- Let $y^{(1)}, \ldots, y^{(m)} \in \mathbb{R}^q$ be the rows of
$$U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} I_q \\ 0 \end{pmatrix}.$$

We will always use $q \leq p$.

Exercise 4.2. In the Definition of PCA, we assumed that the diagonal matrix D satisfied $D_{ii} \geq D_{i+1,i+1}$ for all $1 \leq i \leq p-1$. Show that there always exists a singular value decomposition with this property. That is, if A is an $m \times n$ complex matrix with $m \leq n$, show that there exists an integer $p \leq m$, there exists a diagonal $p \times p$ matrix D with nonnegative values, there exist unitary $m \times m$ U and unitary $n \times n$ V such that
$$A = U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} V,$$
and such that $D_{ii} \geq D_{i+1,i+1}$ for all $1 \leq i \leq p-1$.

Definition 4.3. The map $x^{(i)} \mapsto y^{(i)}$, $1 \leq i \leq m$ from $\mathbb{R}^n$ to $\mathbb{R}^q$ in Algorithm 4.1 could be called a **spectral embedding**. Letting I_q denote the $q \times q$ identity matrix, we could write
$$\begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{pmatrix} = \begin{pmatrix} x^{(1)} \\ \vdots \\ x^{(m)} \end{pmatrix} V^* \begin{pmatrix} I_q \\ 0 \end{pmatrix}$$
(Here we denote $x^{(i)}, y^{(i)}$ as row vectors, for each $1 \leq i \leq m$.) Put another way,
$$y^{(i)} := x^{(i)} V^* \begin{pmatrix} I_q \\ 0 \end{pmatrix}, \qquad \forall\, 1 \leq i \leq m.$$
That is, the map $x^{(i)} \mapsto y^{(i)}$, $1 \leq i \leq m$ from $\mathbb{R}^n$ to $\mathbb{R}^q$ is a linear function. That is, PCA can be understood as a linear function. Note that $V^* \begin{pmatrix} I_q \\ 0 \end{pmatrix}$ is an $n \times q$ matrix. For each

$1 \leq j \leq q$, the j^{th} column of $V^* \begin{pmatrix} I_q \\ 0 \end{pmatrix}$ is called a j^{th} **principal component** of A, or a j^{th} right **singular vector** of A. Since V is unitary, the principal components are orthogonal to each other.

Remark 4.4. In Algorithm 4.1, we could alternatively let $y^{(1)}, \ldots, y^{(m)}$ be the rows of

$$U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix} V,$$

but V only applies a rotation to the data, so including or not including V makes no difference for finding a clustering of our data points. However, in this other definition, the map $x^{(i)} \mapsto y^{(i)}$, $1 \leq i \leq m$ from $\mathbb{R}^n$ to $\mathbb{R}^n$ is a linear projection (i.e. $T^2 = T$). To see this, note that

$$\begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{pmatrix} = \begin{pmatrix} x^{(1)} \\ \vdots \\ x^{(m)} \end{pmatrix} V^* \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix} V$$

(Here we denote $x^{(i)}, y^{(i)}$ as row vectors, for each $1 \leq i \leq m$.) Put another way,

$$y^{(i)} := x^{(i)} V^* \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix} V, \qquad \forall\, 1 \leq i \leq m.$$

Define then $T(x) := xV^* \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix} V$ for all $x \in \mathbb{R}^n$. Then

$$T^2(x) = T(T(x)) = xV^* \begin{pmatrix} I_p & 0 \\ 0 & 0 \end{pmatrix} VV^* \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix} V = xV^* \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix} V = T(x), \qquad \forall\, x \in \mathbb{R}^n.$$

The idea of Algorithm 4.1 is that we would like to choose p so that

$$A \approx U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix} V. \qquad (*)$$

Typically, D will have q large entries with other $p - q$ entries close to zero, in which case $(*)$ holds, since (by Theorem 3.102), $(*)$ reduces to showing that $D \approx D \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix}$.

Although the $\approx$ symbol here is not rigorous, we could make it rigorous with Definition 3.83, Exercise 3.85 and the following Exercise.

Exercise 4.5. Let A be a real $m \times n$ matrix, and define

$$\|A\|_{2\to 2} := \sup_{x \in \mathbb{R}^n : \|x\|_2 \leq 1} \|Ax\|_2 .$$

Let U be an $m \times m$ orthogonal matrix, let V be an $n \times n$ orthogonal matrix, let D be a $p \times p$ diagonal matrix with nonzero entries such that $D_{ii} \geq D_{i+1,i+1}$ for all $1 \leq i < p$. Show that

$$\left\| U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} V - U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} I_q & 0 \\ 0 & 0 \end{pmatrix} V \right\|_{2\to 2} = D_{q+1,q+1}.$$

```
from sklearn import datasets
iris = datasets.load_iris()
data = iris.data
```

The command `data.shape` reveals that `data` is a 150 by 4 matrix. Each row corresponds to a different flower that is measured, and each of the four columns corresponds to four measurements of a flower: sepal length, sepal width, petal length and petal width (all in centimeters). There are three different flower species that are measured in this table: Setosa, Versicolour and Virginica. We will use PCA to try to group the data points (rows of the matrix) into three distinct groups.

```
U, D_vector, V = np.linalg.svd(data)
# D_vector is an array of p = 4 singular values of data
# Let's check that UDV = data
D_truncated = np.zeros([150, 4])
D_truncated[:4, :4] = np.diag(D_vector)
print(np.linalg.norm(  U @ D_truncated @ V  - data))
# This number is small, so U D_truncated V ~ data
```

The singular values of data are approximately: 96, 18, 3 and 2. So, it looks like the largest two singular values capture most of the information about the data, that is,

$$A \approx U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} I_2 & 0 \\ 0 & 0 \end{pmatrix} V.$$

Recall that $A = U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} V$. Recall that U is 150×150, $\begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix}$ is 150 by 4, and V is 4×4. The product

$$U \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} I_2 \\ 0 \end{pmatrix}$$

then is a 150×2 matrix, which we can plot (i.e. we can plot 150 of these two-dimensional vectors in the plane).

```
import matplotlib.pyplot as plt
D_truncated = np.zeros([150, 2])
D_truncated[:2, :2] = np.diag(D_vector[:2])
pca_data = U @ D_truncated
plt.plot(
    pca_data[:, 0],
    pca_data[:, 1],
    'o'
)
plt.xlabel('1st component')
plt.ylabel('2nd component')
plt.savefig('iris1.pdf')
plt.show()

fig, ax = plt.subplots()
ax.scatter(
    pca_data[:, 0],
    pca_data[:, 1],
    c = iris.target
)
plt.xlabel('1st component')
```

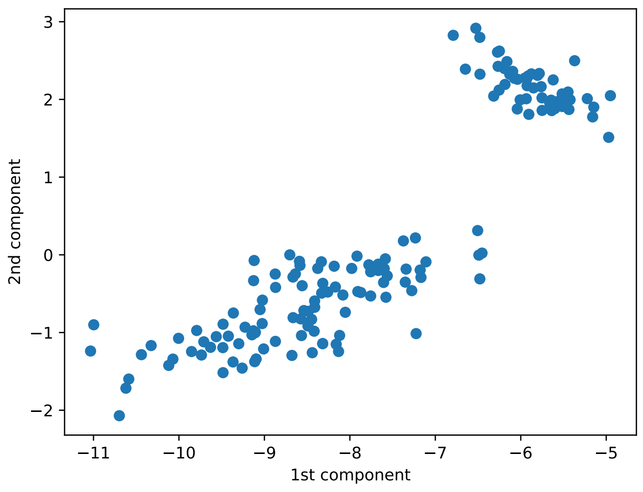


FIGURE 1. Plot of 2-dimensional output of PCA, Iris data set

```
plt.ylabel('2nd component')
plt.savefig('iris2.pdf')
plt.show()
```

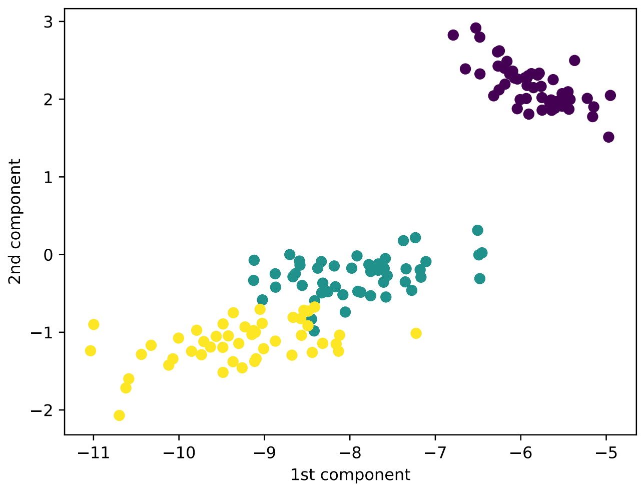


FIGURE 2. Plot of PCA output in two dimensions. Three classes of irises labelled. Iris data set

4.2. **k-Means Clustering.** In the previous section, we used PCA to reduce the dimension of our data. We then tried to identify clusters of data points from two and three dimensional plots. There are various ways to rigorously try to find clusters of data points. One popular method is k-means clustering.

Definition 4.6 (**k-means Clustering**). Let $w^{(1)},\ldots,w^{(m)}\in\mathbb{R}^q$. Let $1\leq k\leq m$ be an integer. Then k-means clustering is the following optimization problem: find a partition $S_1,\ldots,S_k$ of $\{1,\ldots,m\}$ minimizing the quantity

$$\sum_{i=1}^{k}\sum_{j\in S_i}\Big\|w^{(j)}-\frac{1}{|S_i|}\sum_{\ell\in S_i}w^{(\ell)}\Big\|_2^2, \qquad (*)$$

over all such partitions of $S_1,\ldots,S_k$. That is, find the partition of the points $w^{(1)},\ldots,w^{(m)}$ into k clusters that minimizes the above function.

For each $1\leq i\leq k$, the term $\frac{1}{|S_i|}\sum_{y\in S_i}y$ is the center of mass (or barycenter) of the points in S_i, so each term in the sum is the squared distance of some point in S_i from the barycenter of S_i. So, k-means clustering can be seen as a kind of geometric version of least-squares regression. We emphasize that k is fixed.

Remark 4.7. In case S_i is empty for some $1\leq i\leq k$, we define $\frac{1}{|S_i|}\sum_{y\in S_i}y:=0$. However, we may assume that $S_1,\ldots,S_k$ are all nonempty, a priori. To see why, note that if e.g. $S_1=\emptyset$ then $m\geq k$ implies there is some $1\leq t\leq k$ such that $|S_t|\geq 2$. Then, writing $S_t=\{x,\ldots\}$, the partition $\widetilde{S}_1,\ldots,\widetilde{S}_k$ satisfying $\widetilde{S}_r:=S_r$ for all $r\neq 1$, $r\neq t$, $\widetilde{S}_1:=\{x\}$, $\widetilde{S}_t:=S_t\smallsetminus\{x\}$, then

$$\begin{aligned}&\sum_{i=1}^{k}\sum_{j\in\widetilde{S}_i}\Big\|w^{(j)}-\frac{1}{|\widetilde{S}_i|}\sum_{\ell\in\widetilde{S}_i}w^{(\ell)}\Big\|_2^2-\sum_{i=1}^{k}\sum_{j\in S_i}\Big\|w^{(j)}-\frac{1}{|S_i|}\sum_{\ell\in S_i}w^{(\ell)}\Big\|_2^2\\&=\sum_{j\in S_t\smallsetminus\{x\}}\Big\|w^{(j)}-\frac{1}{|S_t|-1}\sum_{\ell\in S_t\smallsetminus\{x\}}w^{(\ell)}\Big\|_2^2-\sum_{j\in S_t}\Big\|w^{(j)}-\frac{1}{|S_t|}\sum_{\ell\in S_t}w^{(\ell)}\Big\|_2^2\end{aligned}$$

We then conclude this quantity is negative by Exercise 4.9, i.e. the quantity $(*)$ is smaller for the partition $\widetilde{S}_1,\ldots,\widetilde{S}_k$.

Exercise 4.8. Let $w^{(1)},\ldots,w^{(m)}\in\mathbb{R}^q$. Let $y\in\mathbb{R}^q$. Show that

$$\sum_{j=1}^{m}\Big\|w^{(j)}-\frac{1}{m}\sum_{\ell=1}^{m}w^{(\ell)}\Big\|_2^2\leq\sum_{j=1}^{m}\Big\|w^{(j)}-y\Big\|_2^2.$$

That is, the barycenter is the point in $\mathbb{R}^q$ that minimizes the sum of squared distances.

Exercise 4.9. Let $w^{(1)},\ldots,w^{(m)}\in\mathbb{R}^q$ with $m\geq 2$. Show that

$$\sum_{i=1}^{m}\Big\|w^{(i)}-\frac{1}{m}\sum_{j=1}^{m}w^{(j)}\Big\|_2^2>\sum_{i=1}^{m-1}\Big\|w^{(i)}-\frac{1}{m-1}\sum_{j=1}^{m-1}w^{(j)}\Big\|_2^2.$$

How can we solve the k-means clustering problem? The most basic algorithm is a "gradient-descent" procedure known as Lloyd's Algorithm.

Algorithm 4.10 (**Lloyd's Algorithm**). Let $w^{(1)},\ldots w^{(m)}\in\mathbb{R}^q$. Choose $z^{(1)},\ldots,z^{(k)}\in\mathbb{R}^q$ (randomly or deterministically), and define $T_i:=\emptyset$ for all $1\leq i\leq k$. Repeat the following procedure:

- For each $1 \leq i \leq k$, re-define
$$T_i := \{j \in \{1, \ldots, m\} \colon \, \|w^{(j)} - z^{(i)}\| = \min_{\ell=1,\ldots,k} \|w^{(j)} - z^{(\ell)}\| \,\}.$$
(If more than one ℓ achieves this minimum, assign j to T_ℓ where ℓ is an arbitrary index ℓ achieving this minimum.) (The sets $T_1, \ldots, T_m$ are called **Voronoi regions**.)
- For each $1 \leq i \leq k$, re-define $z^{(i)} := \frac{1}{|T_i|} \sum_{j \in T_i} w^{(j)}$.

Once this procedure is iterated a specified number of times, output $S_i := T_i$ for all $1 \leq i \leq k$.

Algorithm 4.10 can be considered a "gradient-descent" procedure since the first step of the iteration always decreases the quantity $(*)$ by the definition of $(*)$, and the second step of the iteration always decreases $(*)$ by Exercise 4.8.

While each iteration of Lloyd's Algorithm 4.10 decreases the value of the quantity $(*)$ (non-strictly), iterating this algorithm many times does not guarantee that a global minimum of $(*)$ is found. To see why, recall that the local minimum of a function $f \colon \mathbb{R} \to \mathbb{R}$ may not be the same as a global minimum. So, while Lloyd's Algorithm 4.10 is simple and it might work well in certain situations, it has no general theoretical guarantees, since it will only approach a local minimum of $(*)$ rather than a global minimum. Some work has been done to make a "wise" choice of the initial points $y^{(1)}, \ldots, y^{(k)}$.

So, are there any efficient algorithms with theoretical guarantees? For any $\varepsilon > 0$, there is a $9 + \varepsilon$ factor approximation algorithm for the k-means clustering problem [KMN$^+$04] with a polynomial running time (that does not depend on k). That is, it is possible in polynomial time to find a partition $S_1, \ldots, S_k$ whose value is at most $9 + \varepsilon$ times the minimum possible value of $(*)$. This algorithm is based upon [Mat00]. This factor of 9 was improved to 6.457 in [ANFSW0] and to 6.12903 in [GOR$^+$21], and then to 5.912 in [CAEMN22]. It was shown [ACKS15] that there exists some $\varepsilon > 0$ such that approximating the k-means clustering problem with a multiplicative factor of $1 + \varepsilon$ for all k is NP-hard. This result was improved in [CAS19]. Still, there is a rather large gap between the best general purpose algorithm, and the hardness result. Many algorithms can approximately solve the k-means clustering problem to a multiplicative factor of $1 + \varepsilon$, but these algorithms always have an exponential dependence on k for their run times [HPM04]. So, if we try to use $k = 100$, which occurs in some applications, these algorithms seem to be impractical.

It is possible to combine dimension-reduction techniques (such as PCA or the Johnson-Lindenstrass Lemma, Theorem 5.2) with the above algorithms [CEM$^+$15, MMR18], thereby saving time by working in lower dimensions. However, these techniques do not seem to improve the exponential run times in k.

Some streaming algorithms are known for k-means clustering [Che09, FMS07]. For example, the algorithm of [Che09] uses memory of size $O(q^2 k^2 \varepsilon^{-2} (\log m)^8)$ to approximately solve the k-means problem within a multiplicative factor of $1 + \varepsilon$. Note that the points themselves are not actually stored in this algorithm, otherwise the memory requirement would be at least $\Omega(n)$. In fact, only the barycenters of the clusters are typically stored in these streaming algorithms, which drastically reduces the memory requirement.

Since k-means clustering uses unlabelled data (despite the fact that k needs to be specified), it is considered a method in **unsupervised learning**.

In the code below, I begin with the PCA data from the previous section on the `iris` dataset, and I then run k-means clustering to try to recover the original data clusters (of three iris species).

```
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

ds = datasets.load_iris()
data = ds.data
nrow, ncol = data.shape
q = 2     # number of principal components

U, D_vector, V = np.linalg.svd(data)
print(D_vector)

D_truncated = np.zeros([nrow, q])
D_truncated[:q, :q] = np.diag(D_array[:q])
pca_data = U @ D_truncated
```

Now run k-means on PCA data.

```
k = 3
km = KMeans(n_clusters=k).fit(pca_data)
labels = km.labels_
centers = km.cluster_centers_

if q == 2:
    fig, ax = plt.subplots()
    ax.scatter(
        pca_data[:, 0],
        pca_data[:, 1],
        c = labels.astype(float)
    )
    ax.set_xlabel('1st component')
    ax.set_ylabel('2nd component')
    ax.scatter(centers[:, 0], centers[:, 1], c = 'r')
    plt.show()

if q == 3:
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    #ax.view_init(45, 45, 45)
    ax.scatter(
        pca_data[:, 0],
        pca_data[:, 1],
        pca_data[:, 2],
        c = labels.astype(float)
    )
    ax.set_xlabel('1st component')
    ax.set_ylabel('2nd component')
    ax.set_zlabel('3rd component')
    ax.scatter(
        centers[:, 0],
        centers[:, 1],
        c = 'r'
    )
    plt.show()
```

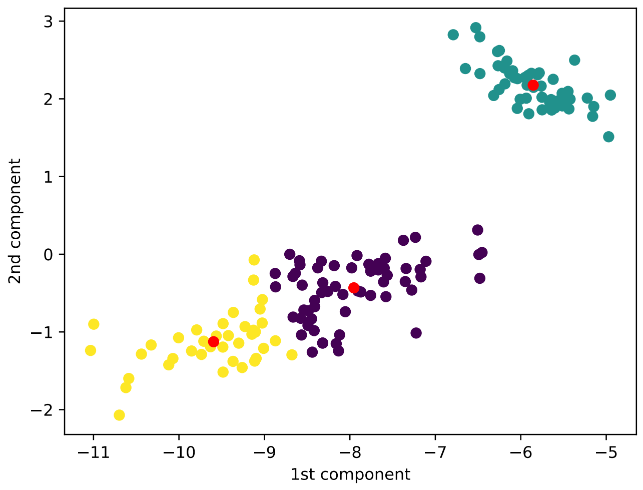


FIGURE 3. Clusters found from k-means with $k = 3$, $q = 2$, with centers. Iris data set. Compare to the true iris data as shown in Figure 2.

Remark 4.11. When I first tried to run this program in Jupyter, I got an Attribute Error, which was fixed by running `pip install threadpoolctl == 3.1.0` in the console.

Exercise 4.12. In this exercise, we will perform some additional analysis on the sklearn built in data sets. For the iris data set, recall that we used PCA to embed the data points (i.e. 150 vectors in $\mathbb{R}^4$) into $\mathbb{R}^2$. We then ran `KMeans` from the sklearn package, with $k = 3$ clusters, and we plotted the resulting k clusters (with k different colors in a scatter plot), along with the k cluster centers

- Add some more information to our previous plot by also plotting the lines between the Voronoi regions. (Hint: if $z^{(1)}, z^{(2)} \in \mathbb{R}^2$ are two centers of two Voronoi regions, then the boundary between the regions is a line that is perpendicular to the straight line between $z^{(1)}$ and $z^{(2)}$, and this line passes through the point $(z^{(1)} + z^{(2)})/2$. This should be enough information to find the equation of this line.)
- Compute the percentage of mis-classified points (e.g. an output of 2% means that only about 2% of data points were mis-classified, i.e. about 98% of data points are correctly clustered).
- Sometimes a dataset is so large, it is difficult to directly use PCA on the data (e.g. the number 150 might be a trillion instead). We can still use PCA though by randomly sampling a small subset of rows of the data matrix, and hopefully performing PCA

on this subset of the data will still be relevant for the full set of data. This statement can be made rigorous, but let's test it out ourselves. Randomly sample 20 rows from the data matrix (using e.g. the `random.sample` command from the `random` package), perform PCA on that resulting dataset, run k-means, and then repeat the above procedure to check for the number of mis-classified points (on the original dataset) that results from the clustering you got from the smaller dataset. (Once you have the cluster centers from the smaller dataset, you can then cluster the larger dataset using the cluster centers from the smaller dataset.) How did the number of mis-classified points change compared to performing PCA on your original dataset?

Repeat the above the for sklearn wine dataset (with $k = 3$), and the sklearn digits dataset (with $k = 10$).

Exercise 4.13. Let A be an $m \times n$ matrix with nonnegative entries. A **nonnegative matrix factorization** for A with k classes is a factorization of the form

$$A = WH,$$

where W is an $m \times k$ matrix, H is a $k \times n$ matrix, and both W, H have nonnegative entries. Sometimes writing a factorization in this way is impossible. (If a factorization exists like this, then A must have rank at most k.) However, we can still try to find W, H that approximately satisfy $WH \approx A$. This is exactly what the Python function `NMF` does (from the `sklearn` package). (More specifically, Python tries to find W, H that minimize a norm of $A - WH$, plus some "regularizing terms". For details, see the sklearn documentation.)

Nonnegative matrix factorization is used in several machine learning applications, e.g. to cluster data into similar groups, in recommendation algorithms, etc. To illustrate this, let's consider the matrix A whose entries are the numbers in the following table

	apple	banana	bell pepper	crab	broccoli	carrot	pear	shrimp
calories	130	110	25	100	45	30	100	100
sodium	0	0	40	330	80	60	0	240
potassium	260	450	220	300	460	250	190	220
carbohydrates	34	30	6	0	8	7	26	0
vitamin A	2	2	4	0	6	110	0	4
vitamin C	8	15	190	4	220	10	10	4

- Verify that A has rank 6, so that we know for sure we cannot write $A = WH$ exactly with $k = 3$.
- Find an approximate nonnegative matrix factorization of A with 3 classes with the Python commands

  ```
  from sklearn.decomposition import NMF
  model = NMF(n_components = 3, init = 'random', random_state = 0)
  W = model.fit_transform(A)
  H = model.components_
  ```

  Do the matrices W, H satisfy $A = WH$? If not, check the value of the norm of $A - WH$ and compare it with the norm of A.
- Each of the three rows of H corresponds to a different class of food. The largest entry in a column of H sorts the food into a given class. For example, the first row of H seems to correspond to "fruits," since the columns for apple, banana, and pear

all have largest values in their top entries. (Carrot also seems to have a largest value here even though it is not a fruit.) What classes of foods do the other two rows of H seem to represent, and which food items are in those classes according to H?

- Each column of W also corresponds to a different class of food (like the rows of H). The largest entry in a row of W indicates which food characteristics are most important for being in each class. For example, calories, potassium, carbohydrates and vitamin A have their largest entries in the first column of W, so these four characteristics are the most significant contributions to being in the class of "fruits" in this table. (Carrot is the only one with a large value of vitamin A so it is unclear why exactly it got sorted in to the class of "fruits.") What food characteristics are most important for the other two classes of foods, according to W?
- When $k = 4$ instead of 3, is the carrot still in the same class as the apple, banana and pear?

Exercise 4.14 (**Finding Topics from Text**). In this exercise, we are going to examine a subset of the `fetch_20newsgroups` dataset from sklearn. This dataset has about 18000 newsgroups posts on 20 topics. This dataset was assembled in 1995 by Ken Lang. (A newsgroup is an online forum that preceded the world wide web.) For simplicity, we will just look at a subset of the dataset covering 6 different topics, as specified below in `categories`.

```
import numpy as np
from sklearn.datasets import fetch_20newsgroups

categories = [
    "comp.graphics",
    "misc.forsale",
    "rec.sport.baseball",
    "sci.space",
    "talk.politics.misc",
    "talk.religion.misc",
]

# import data, remove extraneous bits of text
dataset = fetch_20newsgroups(
    remove = ("headers", "footers", "quotes"),
    subset = "all",
    categories = categories,
    shuffle = True,
    random_state = 42,
)

labels = dataset.target
unique_labels, category_sizes = np.unique(labels, return_counts=True)
true_k = unique_labels.shape[0]

print(f"{len(dataset.data)} documents - {true_k} categories")
```

To get an idea of what the dataset looks like, let's run the command

```
for i in range(3):
    print(dataset.data[i],'\n', dataset.target[i])
```

which prints the first three entries of the dataset, together with their target values.

```
Olympus Stylus, 35mm, pocket sized, red-eye reduction, timer, fully automatic.
Time & date stamp, carrying case.  Smallest camera in its class.
Rated #2 in Consumer Reports.  Excellent condition and only 4 months old.
Worth $169.95.  Purchased for $130.  Selling for $100.
 1

As will I, and the Ultimate Lurker.
 2
I know this has been asked a million time, but..

What was the ftp site carrying 30-40 .ZIPs of full POV "source" files,
including JACK.ZIP and KETTLE.ZIP? I've once been there but
unfortunately lost the address.
I'm in a little hurry with it, so please e-mail me at
jtheinon@kruuna.helsinki.fi. Thanks..
 0
```

That is, the first block of text is in category 1 (miscellaneous for sale items), the next block of text is in category 2 (recreational sports, baseball), and the last block of text is in category 0 (computer graphics). We can view the number of documents in each category with the command `print(category_sizes)`.

In this exercise, we will try to classify the documents correctly. To begin, we will use an "unsupervised" approach, i.e. we will just look at the documents themselves and pretend that we do not know the topic labels. The goal will be to correctly separate the documents into six different categories.

As a first step, we will convert each text document into a sequence of vectors. More specifically, each word in a document will be mapped to a vector. This task can be done with the `CountVectorizer` function.

```
import time as time
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(
    max_df = 0.5,
    min_df = 5,
    stop_words = "english",
)
# eliminate words appearing in more than 50% of documents
# or appearing in less than 5 of the documents

t0 = time.time()
vector_data = vectorizer.fit_transform(dataset.data)

print(f"vectorization done in {time.time() - t0:.3f} s")
print(f"n_samples: {X.shape[0]}, n_features: {X.shape[1]}")
```

With the data vectorized, we can now try to sort the data into six categories using e.g. k-means clustering.

```
k = 6
km = KMeans(n_clusters=k).fit(vector_data)
predicted_labels = km.labels_
centers = km.cluster_centers_
```

Write a program that can check the classification error from this procedure. That is, check how many of the documents are put into the correct group. (For any given permutation of the predicted labels, check the number of labels that agree with the original labels, then take the minimum over all such permutations of the labels $0, 1, 2, 3, 4, 5$.) I got a classification error of around 80%, i.e. only around 20% of the documents are grouped together correctly. This is barely better than a random sorting of around $1 - 1/6 \approx .833$. So, we didn't do very well. Do you get any better performance by changing the parameters .5 and 5? I did not. I thought that some topics might mention certain words exclusively, i.e. maybe some words in the space postings might only appear in the space postings (e.g. "moon"), so if we change .5 to .3, you might expect better performance. However, I did not notice significantly better performance.

Repeat the above procedure using `TfidfVectorizer` instead of `CountVectorizer`. The vectorizer `TfidfVectorizer` will weight the word by its text frequency (the number of times the word appears in one of the newsgroup postings) multiplied by its inverse document frequency (which is typically $1 + \log(1/x)$ where x the number of newsgroup postings where this word appears). In this way, `TfidfVectorzer` will decrease the effect of commonly encountered words such as "a", "an", "the" and so on. It might seem that the .5 or .3 cutoff we used for `CountVectorizer` would have a similar effect. So let's see how `TfidVectorizer` does. I got a classification error of around 45% which is a lot better than before.

Now let's try to "preprocess" the vectorized data matrix using e.g. NMF, and then apply k-means clustering, to see if we can improve our classification.

```
from sklearn.decomposition import NMF
model = NMF(n_components = 6, init = 'random', random_state = 0)
W = model.fit_transform(vector_data)
H = model.components_
```

After applying k-means clustering to W, did you notice any better performance compared to the previous approach?

As a final approach, let's take a "supervised learning" perspective, i.e. we will use an algorithm whose input is labelled data (newsgroup postings with their specified categories), and then using that information we will try to predict the categories of a new batch of newsgroup postings. More specifically, we will use a support vector machine with a kernel (a few more details will be provided on this approach in another exercise).

How does the run time and performance of this approach compare to the previous approaches?

```
from scipy.stats import loguniform
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC
```

```
dataset_train = fetch_20newsgroups(
    remove = ("headers", "footers", "quotes"),
    subset = "train",
    categories = categories,
    shuffle = True,
    random_state = 42,
)

dataset_test = fetch_20newsgroups(
    remove = ("headers", "footers", "quotes"),
    subset = "test",
    categories = categories,
    shuffle = True,
    random_state = 42,
)

combined_data = dataset_train.data + dataset_test.data
labels_train = dataset_train.target

vectorizer = TfidfVectorizer(
    max_df = 0.5,
    min_df = 10,
    stop_words = "english",
)

t0 = time.time()
# we use a vectorizer on the entire dataset, otherwise
# the functions below will output errors
vector_data = vectorizer.fit_transform(combined_data)
print("Vectorized All Data in Time: %0.3f" % (time.time() - t0))
vector_data_train = vector_data[:len(dataset_train.data)]
vector_data_test = vector_data[len(dataset_train.data):]

t0 = time.time()
param_grid = {
    "C": loguniform(1e3, 1e5),
    "gamma": loguniform(1e-4, 1e-1),
}
clf = RandomizedSearchCV(
    SVC(kernel = "rbf", class_weight = "balanced"), param_grid, n_iter=10
)
clf = clf.fit(vector_data_train, labels_train)
print("SVC Search Fit in Time: %0.3f" % (time.time() - t0))
```

```
t0 = time.time()
y_pred = clf.predict(vector_data_test)
true_labels =  dataset_test.target
print("Prediction done in %0.3fs" % (time.time() - t0))
prediction_error = np.sum(y_pred != true_labels) / len(true_labels)

print("prediction error: %0.3f" % prediction_error)
```

Exercise 4.15 (**Eigenfaces**). The goal of facial recognition is to identify the name of someone when given a picture of their face. Suppose we know that our image data set has k distinct faces in it. One way to perform facial recognition is to perform a singular value decomposition on a large amount of facial images. That is, each row of the data matrix corresponds to a facial image, and we apply SVD to the data matrix A. Each facial image must be the same size image (same pixel width and same pixel height). We then write the SVD of A as $A = U\begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix}V$. Then, for any image vector (i.e. for any row vector x representing an image), the dimension reduced image is (recalling Definition 4.3)

$$xV^*\begin{pmatrix} I_q \\ 0 \end{pmatrix}.$$

One way to assign a name to this image x is to apply k-means clustering to the dimension reduced data

$$U\begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} I_q \\ 0 \end{pmatrix},$$

and then check which cluster center the vector $xV^*\begin{pmatrix} I_q \\ 0 \end{pmatrix}$ is closest to. Suppose y is such a cluster center. We then assign a name to x as the most frequently observed name in the cluster associated to y. In this exercise, you will do this procedure on the `lfw_people` dataset in the sklearn package. This data set consists of several different grayscale facial images of famous people; we will only use data from people with at least 70 images in the data set, which amounts to 7 different world leaders from the early 2000s. Below is some code to get you started.

```
import numpy as np
from sklearn.datasets import fetch_lfw_people

lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

# lfw_people.images is a 3-dimensional array, consisting of n_samples of
# images, where each image has width w and height h, in pixels. the
# grayscale value of each image is a real number between 0 and 1
n_samples, h, w = lfw_people.images.shape
train_images = lfw_people.images[100:1+n_samples, :, :]
test_images = lfw_people.images[0:100, :, :]

# the label to predict is the ID of the person
y = lfw_people.target
```

```
target_names = lfw_people.target_names
n_classes = target_names.shape[0]

# put the image data into a 2-dimensional array, where each row of
# the matrix corresponds to a distinct image
train_data = train_images.reshape(train_images.shape[0], -1)
test_data = test_images.reshape(test_images.shape[0], -1)
```

In this code, we have separated the images into 100 test images, and the remaining set of training images. We will perform PCA on the training set of images in order to try to predict the names of the images in the testing set. (Even though we know the identities of the first 100 facial images, we will temporarily assume we do not know their identities.)

To complete this exercise, we do not necessarily need to examine the images, but if you want to see one of them you could use the following code

```
# plot a few images
import matplotlib.pyplot as plt

plt.imshow(lfw_people.images[0].reshape((h, w)), cmap=plt.cm.gray)
```

- Perform PCA on the training data (`train_data`), with $q = 10$ principal components. (Later on we will consider different parameters q). (Do **not** use any built in PCA functions. Just do the PCA yourself.)
- As suggested above, perform k-means clustering on the PCA training data with $k = 7$. Then, predict the label of the first 100 images (`test_data`) using the procedure we described above (assigning the cluster center label that is closest to the image vector). Print out the fraction of correctly classified test images. (I got around 40%, which is just okay, but at least it is better than random assignment, which would get about 14%.) Also report the amount of time it took to perform this entire task, using e.g. the following commands:

  ```
  from time import time
  t0 = time()
  ...  [insert code here]  ...
  print("classification done in %0.3fs" % (time() - t0))
  ```

  (Hint: it might be helpful to use the following Numpy functions; `unique` with `return_counts = True`, and `where`)
- Repeat the previous step with the original training data (`train_data`), rather than the PCA dimension reduced data. (I got the same fraction of correct classification, with about ten times the run time.)
- Using the PCA dimension reduced data (`pca_train_data` and `pca_test_data`), use the code below to try to get a better percentage of correctly classified images. Do this step for $p = 10$ and again for $p = 50$. Did your results improve for $p = 50$?

```
from scipy.stats import loguniform
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC

t0 = time()
```

```
param_grid = {
    "C": loguniform(1e3, 1e5),
    "gamma": loguniform(1e-4, 1e-1),
}
clf = RandomizedSearchCV(
    SVC(kernel = "rbf", class_weight = "balanced"), param_grid, n_iter=10
)
clf = clf.fit(pca_train_data, y[100:1+n_samples])
print("done in %0.3fs" % (time() - t0))
print("Best estimator found by grid search:")
print(clf.best_estimator_)

print("Predicting people's names on the test set")
t0 = time()
y_pred = clf.predict(pca_test_data)
print("done in %0.3fs" % (time() - t0))
number_correct = np.sum(y_pred == true_test_labels)

print("fraction of correct classifications: %0.3f" % (number_correct/100))
print("classification done in %0.3fs" % (time() - t0))
```

- (Optional) Repeat the above where you replace the training data matrix with the mean-subtracted training data matrix. Do your results improve?
- (Optional) Use randomized SVD instead of SVD and compare your results.s
- (Optional) For the SVC option `kernel`, try inputs other than `rbf`, and see if your results improve.

Remark 4.16. As an alternative to k-means clustering and SVC, here is an implementation of kNN (k Nearest Neighbors).

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

t0 = time()
k = 7
clf = Pipeline(
    steps=[
            #("scaler", StandardScaler()),
            ("knn", KNeighborsClassifier(n_neighbors = k))
    ]
)
clf = clf.fit(train_data, y[test_size:1+n_samples])
y_pred = clf.predict(test_data)

number_correct = np.sum(y_pred == true_test_labels)

print("fraction of correct classifications: %0.3f" % (number_correct/test_size))
print("Prediction took", time() - t0, "seconds")
```

For a more detailed view of the success of the classifier, we can print out a report and a confusion matrix. A confusion matric C is defined so that entry C_{ij} is the number of observations known to be in class i and predicted to be in class j (in this case $0 \leq i, j \leq 9$.) So, e.g. $C_{3,9} = 10$ means there are ten examples of digit 3 that were (mistakenly) classified as digit 9.

```
report = metrics.classification_report(
    true_test_labels,
    y_pred,
    digits = 7,
    zero_division = 0
)
confusion = metrics.confusion_matrix(
    true_test_labels,
    test_labels
)
print(
    f"Classification report for classifier:\n"
    f"{report}\n"
    f"{confusion}\n"
)
```

which gave the output

```
Classification report for classifier:
              precision    recall  f1-score   support

           0  0.5000000 0.3125000 0.3846154        16
           1  0.5200000 0.6842105 0.5909091        38
           2  0.6666667 0.3000000 0.4137931        20
           3  0.5619835 0.9066667 0.6938776        75
           4  0.0000000 0.0000000 0.0000000        18
           5  0.0000000 0.0000000 0.0000000        11
           6  0.6250000 0.2272727 0.3333333        22

    accuracy                      0.5500000       200
   macro avg  0.4105214 0.3472357 0.3452184       200
weighted avg  0.4849605 0.5500000 0.4812920       200

[[ 8  0  0  8  0  0  0]
 [19  1  0 18  0  0  0]
 [10  1  0  9  0  0  0]
 [34  4  0 37  0  0  0]
 [10  1  0  7  0  0  0]
 [ 5  0  0  6  0  0  0]
 [14  2  0  6  0  0  0]]
```

In the classification report, the precision is the number of total positives divided by (total positives plus false positives). Recall is the number of total positives divided by (total positives plus false negatives). And f1 score is the harmonic mean of precision and recall. Lastly, support is the number of images in the given class.

Exercise 4.17 (**Classifying Hand Written Digits**)**.** In a previous exercise, we tried to classify handwritten digits from the sklearn built-in digits dataset, using k-means clustering. However, that approach was not very successful. In this exercise taken from the sklearn documentation, we will instead use a support vector machine (SVM)

```
import matplotlib.pyplot as plt
from sklearn import datasets, metrics, svm
from sklearn.model_selection import train_test_split
```

Let's first plot a few of the images to see what they look like. Each image is an 8 by 8 grayscale bitmap, i.e. at 8×8 matrix whose entries have values in $\{0, 1, \ldots, 16\}$.

```
digits = datasets.load_digits()

_, axes = plt.subplots(nrows = 1, ncols = 4, figsize = (10, 3))
for ax, image, label in zip(axes, digits.images, digits.target):
    ax.set_axis_off()
    ax.imshow(image, cmap = plt.cm.gray_r, interpolation = "nearest")
    ax.set_title("Training: %i" % label)
```

As in a previous exercise

```
# flatten the images
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create a classifier: a support vector classifier
clf = svm.SVC(gamma=0.001)

# Split data into 50% train and 50% test subsets
data_train, data_test, labels_train, labels_test = train_test_split(
    data, digits.target, test_size = 0.5, shuffle = False
)
```

For each image vector $x \in \mathbb{R}^{64}$, the `SVC` function by default first embeds that vector x into $\phi(x)$, where $\phi\colon \mathbb{R}^{64} \to \mathbb{R}^n$ for n large, where ϕ satisfies $\langle \phi(x), \phi(y) \rangle = e^{-\gamma \|x-y\|^2}$ for all vectors $x, y \in \mathbb{R}^{64}$, where $\gamma = .001$. (Optional Exericse: show that such a ϕ exists.) Then, for the set of embedded image vectors $\phi(x)$, `SVC` uses a support vector machine to classify the data in the training set. More specifically, for each pair (i, j) of digits $0 \leq i < j \leq 9$, `SVC` chooses a single support vector machine on the training set. (Actually it is impractical to use the embedding ϕ directly. Instead, the SVM is rewritten just in terms of inner products of the form $\langle \phi(x), \phi(y) \rangle = e^{-\gamma \|x-y\|^2}$. The latter function is called a **kernel**. In this way, we only need to consider the kernel itself, i.e. we do not explicitly need to consider the embedding ϕ.)

The **support vector machine** (SVM) is a linear classifier that classifier defined in the following way. Let $x^{(1)}, \ldots, x^{(k)} \in \mathbb{R}^n$ and let $y_1, \ldots, y_k \in \{-1, 1\}$ be given. Assume that there exists $w \in \mathbb{R}^n$ such that

$$\mathrm{sign}(\langle w, x^{(i)}\rangle) = y_i, \qquad \forall\, 1 \leq i \leq k.$$

That is, assume there is a hyperplane perpendicular to w that can classify the vectors $x^{(1)}, \ldots, x^{(k)}$ into two groups (one labelled with $+1$, the other labelled with -1.) (Even without this assumption, an SVM can be defined, but suppose for now that this assumption holds.) The problem is to find the vector w. (We only know that w exists, but we would like an algorithm that finds the vector w.) One way to do this is to use the perceptron algorithm, but that algorithm can only work when the two groups of vectors can be separated by a hyperplane. The SVM is an alternative way to find a vector w that can try to separate the two groups of vectors with a hyperplane, even when no separating hyperplane exists.

The SVM w is defined as follows. Let $\lambda > 0$ and suppose we want to find the $w \in \mathbb{R}^n$ and $z_1, \ldots, z_k \in \mathbb{R}$ minimizing

$$\lambda \|w\|^2 + \frac{1}{k}\sum_{i=1}^{k} z_i.$$

subject to the linear constraints

$$y_i\langle w, x^{(i)}\rangle \geq 1 - z_i, \qquad z_i \geq 0, \quad \forall\, 1 \leq i \leq k.$$

This is a quadratic minimization problem subject to linear constraints, so there are established optimization methods for this task.

To explain what is going on here, consider the quantity

$$\begin{aligned}\theta &:= \min\Big\{\, \|w\| : \forall\, 1 \leq i \leq k,\ y_i\langle w, x^{(i)}\rangle \geq 1\Big\}\\ &= \min\Big\{\, \|w\| : \forall\, 1 \leq i \leq k,\ y_i\Big\langle \frac{w}{\|w\|}, x^{(i)}\Big\rangle \geq \frac{1}{\|w\|}\Big\}.\end{aligned}$$

The quantity $y_i\langle \frac{w}{\|w\|}, x^{(i)}\rangle$ is the distance of vector $x^{(i)}$ from the hyperplane perpendicular to w. So, the vector w minimizing the quantity θ corresponds to the hyperplane through the origin that has the largest uniform distance to the vectors $x^{(1)}, \ldots, x^{(k)}$. Put another way, the **margin** $1/\theta$ measures how wide a symmetric "slab" through the origin can be that separates the vectors $x^{(1)}, \ldots, x^{(k)}$ into their two classes.

```
# Learn the digits on the train subset
clf.fit(data_train, labels_train)

# Predict the value of the digit on the test subset
predicted = clf.predict(data_test)

_, axes = plt.subplots(nrows = 1, ncols = 4, figsize = (10, 3))
for ax, image, prediction in zip(axes, data_test, predicted):
    ax.set_axis_off()
    image = image.reshape(8, 8)
    ax.imshow(image, cmap = plt.cm.gray_r, interpolation = "nearest")
    ax.set_title(f"Prediction: {prediction}")
    print(
    f"Classification report for classifier {clf}:\n"
    f"{metrics.classification_report(labels_test, predicted, digits = 4)}\n"
)
```

First, run the above code. What classification error do you get? Also, if you remove the command `gamma=0.001`, does the classification error improve?

Your task in this exercise is to do adapt the above code to the MNIST dataset. It should be possible to get around 98% correct classification on the test set. (Hint: just use linear SVC, i.e. don't use any kernel method. The above code uses a Gaussian RBF kernel for SVC, since the argument `gamma = 0.001` is used. However, you will probably find that the Gaussian RBF kernel is just too slow when dealing with the 60,000 images in the MNIST dataset.) For MNIST, restrict the training set to the first $k \cdot 1000$ samples for $k = 1, 2, 3, 4, 5$ with both linear SVC and Gaussian RBF SVC with $\gamma = .001$. How does the computation time grow with k?

5. Dimension Reduction

A dimension reduction technique is a way of mapping vectors in a high dimensional space to a much lower dimensional space, while still preserving some structure of the high dimensional vectors. We have already encountered two dimension reduction techniques. The most basic dimension reduction technique is PCA (or more specifically, the spectral embedding from Definition 4.3). Another dimension reduction technique is NMF, which we saw in Exercise 4.13, where the matrix W is a dimension reduced version of the matrix A.

There is another perhaps even more elementary version of dimension reduction, known as Johnson-Lindenstrauss dimension reduction. In this setting, we choose a random projection from $\mathbb{R}^n$ to $\mathbb{R}^{O(\log n)}$ which with high probability almost preserves pairwise distances between a set of n vectors in $\mathbb{R}^n$. There are various ways to use randomness to achieve this goal. We will focus on using a matrix of i.i.d. Gaussians.

5.0.1. *Johson-Lindenstrauss.*

Theorem 5.1 (Concentration of measure for Gaussians, Lipschitz function form)). *Let $f\colon \mathbb{R}^n \to \mathbb{R}$. Suppose that for all $x, y \in \mathbb{R}^n$, $|f(x) - f(y)| \leq \|x - y\|$, so that f is 1-Lipschitz. Let $X = (X_1, \ldots, X_n)$ be a mean zero Gaussian random vector with identity convariance matrix. Then for all $t > 0$,*

$$\mathbf{P}\left(x \in \mathbb{R}^n\colon\ |f(x) - \mathbf{E}f(X)| \geq t\right) \leq 2e^{-2t^2/\pi^2}.$$

Proof. We assume that f all partial derivatives of f exist and are continuous. Let $Y = (Y_1, \ldots, Y_n)$ be another mean zero Gaussian random vector with identity convariance matrix, such that Y and X are independent. Let $0 \leq \theta \leq \pi/2$ and define

$$Z_\theta := X \sin\theta + Y \cos\theta.$$

By rotation invariance of a Gaussian random vector, Z_θ and $\frac{d}{d\theta}Z_\theta = X\cos\theta - Y\sin\theta$ have the same joint distribution as X and Y (since the vectors $(\sin\theta, \cos\theta)$ and $(\cos\theta, -\sin\theta)$ are orthogonal in $\mathbb{R}^2$.) Let $\phi\colon \mathbb{R} \to [0, \infty)$ be a convex function. Using then Jensen's Inequality, then the Chain Rule, then Jensen's inequality and Fubini's Theorem,

$$
\begin{aligned}
&\mathbf{E}\phi(f(X)-\mathbf{E}f(Y)) \leq \mathbf{E}\phi(f(X)-f(Y)) = \mathbf{E}\phi\Big(\int_0^{\pi/2}\frac{d}{d\theta}f(Z_\theta)d\theta\Big)\\
&\quad = \mathbf{E}\phi\Big(\int_0^{\pi/2}\langle(\nabla f)(Z_\theta),\frac{d}{d\theta}Z_\theta\rangle d\theta\Big) = \mathbf{E}\phi\Big(\frac{1}{\pi/2}\int_0^{\pi/2}\frac{\pi}{2}\langle(\nabla f)(Z_\theta),\frac{d}{d\theta}Z_\theta\rangle d\theta\Big)\\
&\quad \leq \mathbf{E}\frac{1}{\pi/2}\int_0^{\pi/2}\phi\Big(\frac{\pi}{2}\langle(\nabla f)(Z_\theta),\frac{d}{d\theta}Z_\theta\rangle\Big)d\theta = \frac{1}{\pi/2}\int_0^{\pi/2}\mathbf{E}\phi\Big(\frac{\pi}{2}\langle(\nabla f)(Z_\theta),\frac{d}{d\theta}Z_\theta\rangle\Big)d\theta\\
&\quad = \frac{1}{\pi/2}\int_0^{\pi/2}\mathbf{E}\phi\Big(\frac{\pi}{2}\langle(\nabla f)(X),Y\rangle\Big)d\theta = \mathbf{E}\phi\Big(\frac{\pi}{2}\langle(\nabla f)(X),Y\rangle\Big)
\end{aligned}
$$

Let $\alpha\in\mathbb{R}$ and let $\phi(x):=e^{\alpha x}$ for all $x\in\mathbb{R}$. Then using independence in Y and Fubini's Theorem,

$$
\mathbf{E}\exp(\alpha[f(X)-\mathbf{E}f(Y)]) \leq \mathbf{E}\exp\Big(\alpha\frac{\pi}{2}\sum_{i=1}^n\frac{\partial f}{\partial x_i}(X)Y_i\Big) = \mathbf{E}_X\prod_{i=1}^n\mathbf{E}_Y\exp\Big(\alpha\frac{\pi}{2}\frac{\partial f}{\partial x_i}(X)Y_i\Big).
$$

Using an explicit computation, for any $s\in\mathbb{R}$ and for any $1\leq i\leq n$,

$$
\mathbf{E}_Y e^{sY_i} = \int_{-\infty}^{\infty}e^{sy}e^{-y^2/2}\frac{dy}{\sqrt{2\pi}} = e^{s^2/2}\int_{-\infty}^{\infty}e^{-(y-s)^2/2}\frac{dy}{\sqrt{2\pi}} = e^{s^2/2}.
$$

So, applying this inequality with $s=\alpha\frac{\pi}{2}\frac{\partial f}{\partial x_i}(X)$ for each $1\leq i\leq n$,

$$
\mathbf{E}\exp(\alpha[f(X)-\mathbf{E}f(Y)]) \leq \mathbf{E}\exp\Big(\alpha^2\frac{\pi^2}{8}\sum_{i=1}^n\Big(\frac{\partial f}{\partial x_i}(X)\Big)^2\Big) \leq \exp\Big(\alpha^2\frac{\pi^2}{8}\Big).
$$

(Since f is 1-Lipschitz, $|\langle\nabla f(x),y\rangle|\leq 1$ for all $x,y\in\mathbb{R}^n$ with $\|y\|\leq 1$. In particular, using $y:=\nabla f(x)/\|\nabla f(x)\|$, we get $\|\nabla f(x)\|\leq 1$.) So,

$$
\begin{aligned}
\mathbf{P}(f(X)-\mathbf{E}f(Y)>t) &= \mathbf{P}(\exp(\alpha[f(X)-\mathbf{E}f(Y)])>e^{\alpha t})\\
&\leq e^{-\alpha t}\exp\Big(\alpha^2\frac{\pi^2}{8}\Big) = \exp\Big(-\alpha t+\alpha^2\frac{\pi^2}{8}\Big).
\end{aligned}
$$

The minimum α occurs when $\alpha=4t/\pi^2$, so making this choice of α, we get

$$
\mathbf{P}(f(X)-\mathbf{E}f(Y)>t)\leq\exp(-2t^2/\pi^2).
$$

Similarly, $\mathbf{P}(f(X)-\mathbf{E}f(Y)<-t)\leq\exp(-2t^2/\pi^2)$, so that

$$
\begin{aligned}
\mathbf{P}(|f(X)-\mathbf{E}f(Y)|>t) &= \mathbf{P}(f(X)-\mathbf{E}f(Y)>t)+\mathbf{P}(f(X)-\mathbf{E}f(Y)<-t)\\
&\leq 2\exp(-2t^2/\pi^2).
\end{aligned}
$$

□

Theorem 5.2 (Johnson-Lindenstrauss Lemma). *Let $x^{(1)},\ldots,x^{(m)}\in\mathbb{R}^n$. Let $\varepsilon>0$. Then there exists a linear function $h\colon\mathbb{R}^n\to\mathbb{R}^{\lceil 2^{12}\varepsilon^{-2}\log m\rceil}$ such that*

$$
\left\|x^{(i)}-x^{(j)}\right\| \leq \left\|h(x^{(i)})-h(x^{(j)})\right\| \leq (1+\varepsilon)\left\|x^{(i)}-x^{(j)}\right\|,\qquad \forall\, 1\leq i,j\leq m.
$$

One proves this via the probabilistic method. By concentration of measure, a random projection does what we require.

Proof. Fix $1 \leq k \leq m$. Let $\Pi\colon \mathbb{R}^m \to \mathbb{R}^m$ be the orthogonal projection such that

$$\Pi(z_1, \ldots, z_m) := (z_1, \ldots, z_k, 0, \ldots, 0), \qquad \forall\,(z_1, \ldots, z_m) \in \mathbb{R}^m.$$

Let $X = (X_1, \ldots, X_m)$ be a standard m-dimensional Gaussian random vector. Define

$$a := \mathbf{E}\,\|\Pi X\|\,.$$

We will eventually show that $a \geq 10^{-2}\sqrt{k}$. Observe

$$\mathbf{E}\,\|\Pi X\|^2 = \mathbf{E}\sum_{i=1}^{k} X_i^2 = k\mathbf{E}X_1^2. = k. \qquad (*)$$

Now, by Theorem 5.1 for the 1-Lipschitz function $x \mapsto \|\Pi x\|$,

$$\begin{aligned}
\mathbf{E}\,\|\Pi X\|^4 &= \int_0^\infty 4u^3\mathbf{P}(\|\Pi X\| \geq u)du \\
&= \int_0^{2a} 4u^3\mathbf{P}(\|\Pi X\| \geq u)du + \int_{2a}^\infty 4u^3\mathbf{P}(\|\Pi X\| \geq u)du \\
&\leq \int_0^{2a} 4u^3 du + \int_{2a}^\infty 4u^3\mathbf{P}(|\,\|\Pi X\| - a| > u/2)du \\
&\leq 16a^4 + 8\int_{2a}^\infty u^3 e^{-u^2/2\pi^2}du = 16a^4 + 8(2\pi^2)(2a^2+\pi^2)e^{-2a^2/\pi^2} \leq 16a^4 + 2\pi^4 \\
&\leq 16a^4 + 200k^2 \leq 216\left(\int_{\mathbb{R}^m} \|\Pi x\|^2\,\gamma_m(x)dx\right)^2 \quad \text{, using Jensen's inequality and } (*).
\end{aligned}$$

So, if $Z := \|\Pi X\|$ is a random variable, we have shown that $\mathbf{E}Z^4 < c(\mathbf{E}Z^2)^2$ where $c := 216$. So, using Hölder's Inequality, for $p = 3/2$, $q = 3$,

$$\mathbf{E}Z^2 = \mathbf{E}(Z^{2/3}Z^{4/3}) \leq (\mathbf{E}Z)^{2/3}(\mathbf{E}Z^4)^{1/3} \leq (\mathbf{E}Z)^{2/3}c^{1/3}(\mathbf{E}Z^2)^{2/3}.$$

Using this inequality and $(*)$,

$$\mathbf{E}Z \geq c^{-1/2}\sqrt{\mathbf{E}Z^2} \geq 216^{-1/2}\sqrt{k}. \qquad (**)$$

In summary, $a \geq 2^{-4}\sqrt{k}$ for a defined above.

Let A be an $m \times m$ matrix of i.i.d. standard Gaussian random variables. Fix $x^{(0)} \in \mathbb{R}^m$ with $\|x\| = 1$. By rotation invariance of the Gaussian measure, A and AQ have the same distribution where Q is a fixed $m \times m$ orthogonal matrix, so if we choose Q so that $Q(1, 0, \ldots, 0)^T = x^{(0)}$, we get

$$\begin{aligned}
\mathbf{P}\left(A \in \mathbb{R}^{m\times m}\colon\ \left|\,\left\|\Pi A x^{(0)}\right\|_2 - a\right| \geq \varepsilon a\right) &= \mathbf{P}\left(A \in \mathbb{R}^{m\times m}\colon\ \left|\,\left\|\Pi A(1, 0, \ldots, 0)^T\right\|_2 - a\right| \geq \varepsilon a\right) \\
&= \mathbf{P}\left(X \in \mathbb{R}^m \,|\, \|\Pi X\| - a| \geq \varepsilon a\right).
\end{aligned}$$

So, by Theorem 5.1 applied to the 1-Lipschitz function $x \mapsto \|\Pi x\|$, and using $a \geq 2^{-4}\sqrt{k}$, for any $\varepsilon > 0$, and for any

$$\mathbf{P}\left(A \in \mathbb{R}^{m\times m}\colon\ \left|\,\left\|\Pi A x^{(0)}\right\|_2 - a\right| \geq \varepsilon a\right) \leq 2e^{-2\varepsilon^2 a^2/\pi^2} \leq 2e^{-2^{-10}k\varepsilon^2}.$$

Let $x^{(1)}, \ldots, x^{(n)}$ be n points in $\mathbb{R}^m$. If $k \geq 2^{12}\varepsilon^{-2}\log n$, the union bound shows that

$$\mathbf{P}\left(A \in \mathbb{R}^{m\times m}\colon \exists\, i \neq j\colon\ \left|\,\left\|\Pi A\left(\frac{x^{(i)} - x^{(j)}}{\|x^{(i)} - x^{(j)}\|}\right)\right\| - a\right| \geq \varepsilon a\right) \leq \binom{n}{2}2e^{-2^{-10}k\varepsilon^2} < 1.$$

For any $1 \leq i \leq n$, define $y_i := \Pi A x^{(i)}/(a(1-\varepsilon))$. Then $\exists\, A \in \mathbb{R}^{n\times m}$ such that

$$1 \leq \left\| \frac{y^{(i)} - y^{(j)}}{\|x^{(i)} - x^{(j)}\|} \right\| \leq \frac{1+\varepsilon}{1-\varepsilon} \leq 1 + 3\varepsilon, \qquad \forall\, 1 \leq i, j \leq n.$$

So, our required embedding is $h := \frac{\Pi A}{a(1-\varepsilon)}$, so that $h(x^{(i)}) = y^{(i)}$ for all $1 \leq i \leq n$. Note that h is linear and its nonzero entries form a rectangular matrix of i.i.d. Gaussians. Also, we can choose $k := \lceil 2^{12}\varepsilon^{-2}\log n \rceil$. (In fact, if we choose k to be slightly larger, then the probability becomes exponentially small, so essentially all A satisfies our desired property, hence essentially all linear projections $h\colon \mathbb{R}^n \to \mathbb{R}^{O(\varepsilon^{-2}\log n)}$ satisfy our desired property.) $\square$

Remark 5.3. The Johnson-Lindenstrauss projection h can be implemented in Python with the `GaussianRandomProjection` function, with optional parameter `eps` (the same as the parameter ε from Theorem 5.2), where the dimension of the range of the linear function h is automatically computed.

```
import numpy as np
from sklearn import random_projection
x = np.random.rand(100, 10000)
#projection = random_projection.GaussianRandomProjection()
projection = random_projection.GaussianRandomProjection(eps = .5)
#projection = random_projection.SparseRandomProjection()
#projection = random_projection.SparseRandomProjection(eps = .5)

x_projected = projection.fit_transform(x)
print(x_projected.shape)
```

This code has output

```
(100, 221)
```

That is, we asked for $\varepsilon = 1/2$ in Theorem 5.2, and we reduced the dimension of the row vectors of x from 10,000 to 221.

We can then check for agreement with Theorem 5.2. For example,

```
np.linalg.norm(x[0, :] - x[1, :])
```

outputs

```
40.57920880597641
```

and

```
np.linalg.norm(x_projected[0, :] - x_projected[1, :])
```

outputs

```
40.265214331165566
```

In particular, the ratio of the distance between the first two rows of x and the projected first two rows of x is much smaller than the $3/2$ guarantee from Theorem 5.2.

We also included a commented out code for `SparseRandomProjection()`, which uses a sparse linear projection (a projection matrix with -1 and 1 entries but mostly zeros) instead of i.i.d. Gaussians as in Theorem 5.2.

Exercise 5.4. High-dimensional geometry is much different than low-dimensional geometry, as this exercise demonstrates.

- Show that "most" of the mass of an n-dimensional Gaussian is concentrated on the sphere of radius $\sqrt{n}$ centered at the origin. That is, if $X_1, \ldots, X_n$ are n i.i.d. standard Gaussian random variables, then
$$\lim_{n\to\infty} \mathbf{P}(\sqrt{X_1^2 + \cdots + X_n^2} \in (n + \sqrt{3n}, n - \sqrt{3n}) \geq 2/3.$$
In fact, you should be able to compute the limit exactly.
- Generally, "most" of the mass of a high-dimensional convex body is concentrated near the surface of the body. Let Vol_n denote the usual volume in $\mathbb{R}^n$ (so that the volume of a unit square $[0,1]^n$ is 1.) For example, show that, for any $\varepsilon > 0$,
$$\lim_{n\to\infty} \mathrm{Vol}_n\Big(\Big[-\frac{1}{2}(1-\varepsilon), \frac{1}{2}(1-\varepsilon)\Big]^n\Big) = 0.$$
- Let $B_n := \{x \in \mathbb{R}^n \colon \|x\| \leq 1\}$ be the unit ball centered at the origin. Show that
$$\lim_{n\to\infty} \mathrm{Vol}_n(B_n) = 0.$$
- Let $C_n = \{x \in \{[-1/2, 1/2]^n \colon \exists\, y \in \{-1/2, 1/2\}^n \text{ such that } \|x - y\| \leq 1/2\}\}$ be the union of balls of radius $1/2$ centered at the corners of the hypercube $[-1/2, 1/2]^n$. Let $D_n := \{x \in \mathbb{R}^n \colon \|x\| \leq r\}$ be a ball of radius r centered at the origin, where r is chosen to be as large as possible so that D_n does not intersect the interior of C_n. (Put another way, D_n is tangent to the balls C_n.) Find
$$\lim_{n\to\infty} \mathrm{Vol}_n(D_n).$$
Before you do the computation, try to guess what the answer should be.

6. Pandas

6.1. **Series, DataFrames.**

6.1.1. *Series.* In Pandas, a `Series` is a length n array of objects, together with a length n array of objects called an index. By default, the index is set to $0, 1, \ldots, n-1$. For example, the command `obj = pd.Series([6, 7, 5, -9])` produces the following output

```
0   6
1   7
2   5
3   -9
dtype: int64
```

The command `obj.array` returns `[6, 7, 5, -9]` and the command `obj.index` returns `RangeIndex(start = 0, stop = 4, step = 1)` . The Series array can be called using standard syntax, so that `obj[1]` outputs 7 and `obj[1:3]` outputs

```
1   7
2   5
dtype: int64
```

Part of the flexibility of Pandas is allowing indices other than ordered integers. For example, the command

```
obj = pd.Series([6, 7, 5, -9], index = ["d", "c", "a", "b"])
```

produces the following output

```
d   6
c   7
a   5
b  -9
dtype: int64
```

Similar to before, the Series array elements can be queried, so that `obj["c"]` outputs 7 and `obj[["c", "a"]]` outputs

```
c   7
a   5
dtype: int64
```

Applying functions to the Series will maintain the index of the Series. For example, `obj - 3` outputs

```
d   3
c   4
a   2
b  -12
dtype: int64
```

Also, the index can be reassigned:

```
obj.index= ["x", "y", "z", "w"]
```

outputs

```
x   3
y   4
z   2
w  -12
dtype: int64
```

Recalling Section 1.2, a dictionary of the following form

```
sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}
```

can be converted to a Series with the command `obj = pd.Series(sdata)`, which outputs

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

A Series can then be converted back to a dictionary: `obj.to_dict()` outputs

```
{'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

The index of the Series from `sdata` can be rearranged in the following way.

```
states = ["California", "Ohio", "Oregon", "Texas"]
obj2 = pd.Series(sdata, index = states)
```

which outputs

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Since `sdata` has no index value for "California", the Series value was set to `NaN`. We can check for NaN values using the commands `pd.isna(obj2)` or `obj2.isna()`, which output

```
California      True
Ohio            False
Oregon          False
Texas           False
dtype: bool
```

Similarly, `pd.notna(obj2)` and `obj2.notna()` are the negation of the Series above. (We can create a `NaN` value using e.g. `np.nan`.)

Arithmetic operations can be applied to Series, in a way that respects the indices. For example, `obj + obj2` outputs

```
California      NaN
Ohio            70000.0
Oregon          32000.0
Texas          142000.0
Utah              NaN
```

Both the Series itself and its index can be assigned names. For example

```
obj2.name = "population"
obj2.index.name = "state"
```

results in `obj2` having the form

```
state
California     NaN
Ohio           35000.0
Oregon         16000.0
Texas          71000.0
Name: population, dtype: float64
```

6.1.2. *DataFrame.* A DataFrame is a dictionary of Series, where all of the Series have the same index. A DataFrame can be visualized as a matrix, where each column has a name, and each row corresponds to an index value. You might think of a DataFrame as similar to an Excel spreadsheet. For example,

```
data = {
    "state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
    "year": [2000, 2001, 2002, 2001, 2002, 2003],
    "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]
}
frame = pd.DataFrame(data)
```

outputs

```
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

Specifying a sequence of columns can reorder those columns:

```
frame2 = pd.DataFrame(data, columns=["year", "state", "pop"])
```

outputs

```
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

The command `frame2["state"]` and `frame2.state` both display the "state" column of the above DataFrame, as a Series:

```
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object
```

However, a command of the latter form `frame2.(...)` will not output a Series contained in `frame2` if a built-in DataFrame function conflicts with the column name of the DataFrame, or if the column name contains whitespace or special characters besides underscore.

Rows of a DataFrame can also be displayed as Series. For example, `frame2.loc[3]` and `frame2.iloc[3]` both output

```
year       2001
state    Nevada
pop         2.4
Name: 3, dtype: object
```

Columns of a DataFrame can be assigned values, e.g. `frame2["pop"] = 3` results in `frame2` taking the form

```
   year   state  pop
0  2000    Ohio  3
1  2001    Ohio  3
2  2002    Ohio  3
3  2001  Nevada  3
4  2002  Nevada  3
5  2003  Nevada  3
```

Then `frame2["pop"] = np.arange(6)` results in `frame2` taking the form

```
   year   state  pop
0  2000    Ohio  0
1  2001    Ohio  1
2  2002    Ohio  2
3  2001  Nevada  3
4  2002  Nevada  4
5  2003  Nevada  5
```

(An error would result from the assignment `frame2["pop"] = np.arange(5)` .) The assignment `frame2["temp"] = 3 + np.arange(6)` creates a new column on the right side:

```
   year   state  pop  temp
0  2000    Ohio  0       3
1  2001    Ohio  1       4
2  2002    Ohio  2       5
3  2001  Nevada  3       6
4  2002  Nevada  4       7
5  2003  Nevada  5       8
```

Assigning a column to be equal to a Series will fill in values according to the index, leaving unassigned values as `NaN`. So,

```
frame2["temp"] = pd.Series([5, 7, 8] , index = [3, 2, 5])
```

results in

```
   year   state  pop  temp
0  2000    Ohio  0   NaN
1  2001    Ohio  1   NaN
2  2002    Ohio  2   6
3  2001  Nevada  3   5
4  2002  Nevada  4   NaN
5  2003  Nevada  5   8
```

The “temp” column can be deleted with the command

```
del frame2["temp"]
```

As with NumPy arrays, changing a subset of values from a DataFrame will also change the DataFrame itself. For example

```
y = frame2["state"]
y[2] = "Hawaii"
```

will change the corresponding value of `frame2`, even though we only changed the entry of `y` . (For this reason, Jupyter gave me a warning when I did these commands.) Again by analogy with NumPy, a DataFrame can be transposed with the command `frame2.T` . However, transposing the DataFrame will discard the data types of the columns, unless they all have the same data type.

We can also create a DataFrame using a dictionary of dictionaries:

```
populations = {
    "Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6},
    "Nevada": {2001: 2.4, 2002: 2.9}
}
frame3 = pd.DataFrame(populations)
```

outputs

```
       Ohio  Nevada
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9
```

Note that the indices of the “Ohio” and “Nevada” Series were combined. We can also specify the indices we want the DataFrame to have as follows.

```
pd.DataFrame(populations, index = [2000, 2001, 2003])
```

outputs

```
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
2003   NaN     NaN
```

WARNING. Accessing individual entries of a DataFrame has the opposite ordering convention of NumPy arrays. That is

```
frame2["state"][2]
```

outputs "Ohio", whereas `frame2[2]["state"]`, which mimics NumPy syntax, will produce an error. However, `frame2.T["state"][2]` will output "Ohio".

A DataFrame's index has a name attribute, and a DataFrame's columns have a (single) name attribute.

```
frame3.index.name = "year"
frame3.columns.name = "state"
```

outputs

```
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9
```

However, the DataFrame itself does not have a name attribute.

A DataFrame can be cast as a NumPy array with the command

```
frame3.to_numpy()
```

The index of a Series or DataFrame can be accessed as

```
ind = frame3.index
```

with output

```
Index([2000, 2001, 2002], dtype = 'int64', name = 'year')
```

An Index object cannot be changed with the equality symbol, but it can be changed with methods such as `append`, `union`, and so on. For example,

```
ind.append(pd.Index(["2007"]))
```

outputs

```
Index([2000, 2001, 2002, '2007'], dtype = 'object')
```

6.2. **Reindexing, Deletion.** It is often useful to rearrange the index of a Series or DataFrame via the `reindex` function, as follows. Recall that

```
obj = pd.Series([6, 7, 5, -9], index = ["d", "c", "a", "b"])
```

produces the following output

```
d   6
c   7
a   5
b  -9
dtype: int64
```

Then

```
obj2 = obj.reindex(["a", "b", "c", "d", "e"])
```

produces the following output

```
a   5.0
b   -9.0
c   7.0
d  6.0
e  NaN
dtype: float64
```

Note that `obj` itself is unchanged from the `obj.reindex` command.

If an Index consists of a strictly increasing sequence of integers, then the `reindex` option `method = "ffill"` can fill in missing Series entries by copying preceding entries, in the following way.

```
obj3 = pd.Series(["blue", "red", "green"], index = [0, 2, 4])
obj4 = obj3.reindex(np.arange(6), method = "ffill")
```

produces the following output for `obj4`

```
0    blue
1    blue
2    red
3    red
4    green
5    green
dtype: object
```

The options `method = "bfill"` and `method = "nearest"` similarly fill in missing data. If the Index does not consist of a strictly increasing sequence of integers, then the `reindex` command will round and delete some index entries to a new integer valued Index.

The Index or the columns of a DataFrame can be reindexed. Recall that

```
data = {
    "state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
    "year": [2000, 2001, 2002, 2001, 2002, 2003],
    "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]
}
frame = pd.DataFrame(data)
```

outputs

```
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

Then

```
frame2 = frame.reindex(index = [3, 2, 5])
```

outputs

```
    state  year  pop
3  Nevada  2001  2.4
2    Ohio  2002  3.6
5  Nevada  2003  3.2
```

Note that we were able to delete rows from the DataFrame by removing those index values from the `reindex` function input. Also

```
frame3 = frame.reindex(columns = ["pop", "state", "month"])
```

outputs

```
   pop  state   month
0  1.5  Ohio    NaN
1  1.7  Ohio    NaN
2  3.6  Ohio    NaN
3  2.4  Nevada  NaN
4  2.9  Nevada  NaN
5  3.2  Nevada  NaN
```

The `drop` function can delete entries from a Series or rows/columns from a DataFrame. For example

```
frame.drop(index = [0, 1, 2])
```

outputs

```
    state  year  pop
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

The command `frame.drop([0, 1, 2], axis = 0)` outputs the same result.

Similarly,

```
frame.drop(columns = ["state", "pop"])
```

outputs

```
   year
0  2000
1  2001
2  2002
3  2001
4  2002
5  2003
```

The command `frame.drop(["state", "pop"], axis = 1)` outputs the same result.

A column of the DataFrame can be set as the index. For example

```
frame2.set_index("year")
```

outputs

```
              state        pop
year
2000          Ohio         1.5
2001          Ohio         1.7
2002          Ohio         3.6
2001          Nevada       2.4
2002          Nevada       2.9
2003          Nevada       3.2
```

This and other functions can make permanent changes to the DataFrame with the added argument `inplace = True`, e.g.

```
frame2.set_index("year", inplace = True)
```

We could even make a multi-index (i.e. an index with more than one argument) with the command

```
frame3 = pd.DataFrame(data, columns=["year", "state", "pop"])
frame3.set_index(["year", "state"], inplace = True)
```

which outputs

```
                                pop
year          state
2000          Ohio         1.5
2001          Ohio         1.7
2002          Ohio         3.6
2001          Nevada       2.4
2002          Nevada       2.9
2003          Nevada       3.2
```

Then

```
frame3.loc[2002, "Ohio"]
```

outputs

```
pop    3.6
Name: (2002, Ohio), dtype: float64
```

and

```
frame3.iloc[3]
```

outputs

```
pop    2.4
Name: (2001, Nevada), dtype: float64
```

As with Numpy arrays, changes to a sub-object of a DataFrame will be inherited by the original DataFrame. For example,

```
y = frame2["state"]
y[4] = "Nvidia"
frame2
```

outputs

```
        year     state        pop
0       2000     Ohio         1.5
1       2001     Ohio         1.7
2       2002     Ohio         3.6
3       2001     Nevada       2.4
4       2002     Nvidia       2.9
5       2003     Nevada       3.2
```

A DataFrame can be converted to a Numpy array with the commands `frame2.values` or `frame2.to_numpy`. The former command deletes the column names and index.

Analogous functions can be applied to Series:

```
obj3 = pd.Series(["blue", "red", "green"], index = [0, 2, 4])
obj4 = obj3.drop([0, 2])
```

produces the following output for `obj4`

```
4    green
dtype: object
```

Remark 6.1. A newly developed Python package Polars is often faster to use than Pandas, though Polars has no indices, i.e. it is similar to using Pandas where every index is of the form $0, 1, 2, \ldots$. Some syntax in Polars is similar to Pandas, though other syntax is quite different. Also, in Polars a DataFrame is immutable, unlike Pandas where a DataFrame is mutable.

Exercise 6.2. This exercise deals with sunspot data from the following files (the same data appears in different formats)

txt file   csv file

These files are taken from http://www.sidc.be/silso/datafiles#total

To work with this data, e.g. with pandas in Python you can use the command

```
df = pd.read_csv('SN_d_tot_V2.0.csv')
```

to import the .csv file.

The format of the data is as follows.

- Columns 1-3: Gregorian calendar date (Year, Month, then Day)
- Column 4: Date in fraction of year
- Column 5: Daily total number of sunspots observed on the sun. A value of -1 indicates that no number is available for that day (missing value).
- Column 6: Daily standard deviation of the input sunspot numbers from individual stations.
- Column 7: Number of observations used to compute the daily value.
- Column 8: Definitive/provisional indicator. A blank indicates that the value is definitive. A '*' symbol indicates that the value is still provisional and is subject to a possible revision (Usually the last 3 to 6 months)

It is known that the number of sunspots on the sun follows an approximately 11-year sinusoidal pattern. So, if we plot the number of sunspots over several years, the distance between the highest observed numbers of sunspots should be around 11 years.

Let U_t be the number of sunspots at time t, where t is measured in years. We model U_t as

$$U_t = m_t + a\cos(2\pi\theta t) + b\sin(2\pi\omega t)) + Y_t, \qquad \forall\, t \in \mathbb{R},$$

where $a, b, \theta, \omega \in \mathbb{R}$ are unknown (deterministic) parameters, m_t is an unknown deterministic function of t that is assumed to be a "slowly varying" function of t, and $\{Y_t\}_{t\in\mathbb{R}}$ are i.i.d. mean zero random variables. The quantity m_t is called the **trend** and the quantity $s_t := a\cos(2\pi\theta t) + b\sin(2\pi\omega t))$ is called the **seasonal component** of the time series $\{U_t\}_{t\in\mathbb{R}}$.

Since the 11-year sinusoidal pattern is known, we assume for now that $\theta = \omega = 1/11$. Note that

$$\sum_{s=t,t+1/365,t+2/365,\ldots,t+11} \cos(2\pi\theta s) \approx 0, \qquad \sum_{s=t,t+1/365,t+2/365,\ldots,t+11} \cos(2\pi\theta s) \approx 0, \qquad \forall\, t \in \mathbb{R}.$$

So, if m_t is slowly varying in the sense that $m_t \approx \frac{1}{11\cdot 365.25}\sum_{s=t-5.5,t-5.5+1/365,t+2/365,\ldots,t+5.5} m_s$, an unbiased estimator for m_t is

$$M_t := \frac{1}{11\cdot 365.25}\sum_{s=t-5.5,t-5.5+1/365,t+2/365,\ldots,t+5.5} U_s.$$

M_t defined in this way is called a **moving average**.

• Since -1 denotes a missing data value, we should first consider how to fill in missing data values. Let's first use the `ffill` option of `reindex` to fill in these missing values. (Since `ffill` works best when the index consists of increasing integers, you should either convert the first three column entries of a row to a single integer, or you could take the fourth column entry and multiply it by 1000 to get an integer.)

• Plot M_t versus t. Do you observe any fluctuations in M_t or does it seem to be roughly constant? If so, what is this constant?

Once we have the estimate M_t, we can then use the approximation

$$U_t - M_t \approx a\cos(2\pi\theta t) + b\sin(2\pi\omega t)) + Y_t, \qquad \forall\, t \in \mathbb{R},$$

and then try to estimate a, b. A general way to estimate $s_t := a\cos(2\pi\theta t) + b\sin(2\pi\omega t))$ is to use a (smaller) moving average such as

$$S_t := \frac{1}{11}\sum_{s=t-5/365,t-4/365,\ldots,t+5/365} [U_s - M_s].$$

Note that S_t is unbiased.

• Plot S_t versus t. Does it look like a sinusoidal curve? Note that S_t removed the trend from the time series.

Another way to estimate s_t is to estimate the constants a and b directly. By the double angle formula, note that

$$\sum_{s=t,t+1/365,t+2/365,\ldots,t+11} \cos(2\pi\theta s)\sin(2\pi\theta s) = \sum_{s=t,t+1/365,t+2/365,\ldots,t+11} \frac{1}{2}\sin(4\pi\theta s) \approx 0.$$

Also,

$$\frac{1}{365.25}\sum_{s=t,t+1/365,t+2/365,\ldots,t+11} \cos^2(2\pi s/11) \approx \int_0^{11} \cos^2(2\pi x/11)dx \approx 11/2.$$

So, an unbiased estimator for a is

$$A_t := \frac{2}{11\cdot 365.25}\sum_{s=t,t+1/365,t+2/365,\ldots,t+11} (U_s - M_s)\cos(2\pi\theta s), \qquad \forall\, t \in \mathbb{R}.$$

Similarly, an unbiased estimator for b is

$$B_t := \frac{2}{11 \cdot 365.25} \sum_{s=t,t+1/365,t+2/365,\ldots,t+11} (U_s - M_s)\sin(2\pi\theta s), \qquad \forall\, t \in \mathbb{R}.$$

• Plot A_t versus t. Plot B_t versus t. Are they close to being constant in t?

• Plot $U_t - [M_t + A_t\cos(2\pi t/11) + B_t\sin(2\pi t/11))]$ versus t. This is the time series with the trend and seasonal components removed. Does this plot "resemble" a stationary process?

• Our modeling assumptions used a period of 11 for the seasonal component of the time series. Does the data reflect this assumption? For example, would it be more accurate to have $\theta = \omega = 1/(10.9)$ in our modeling assumption?

6.3. **Selection, Filtering.** It is preferred to select entries of Series and DataFrames using the `.loc` or `.iloc` functions as follows.

```
obj = pd.Series([6, 7, 5, -9], index = ["d", "c", "a", "b"])
obj2 = pd.Series([6, 7, 5, -9], index = [2, 3, 4, 1])
```

Then

```
obj.iloc[[1, 2]]
```

produces the following output

```
c    7
a    5
dtype: int64
```

The command `obj.loc[["c", "a"]]` produces the same output. Similarly,

```
obj2.iloc[[1, 2]]
```

produces the following output

```
3    7
4    5
dtype: int64
```

The command `obj.loc[[3, 4]]` produces the same output. Note that the syntex here **does not use parentheses**, unlike other functions in Python.

Here `iloc` outputs entries of the Series as if it were a Numpy array (i.e. with an index of the form $0, 1, 2, \ldots$), and `loc` outputs the entries of the Series according to the queried Index values. Furthermore, `obj.iloc[-1]` outputs -9, i.e. we can use negative indices as in Numpy arrays.

The functions `loc` and `iloc` are preferred over simpler index calls since `obj` and `obj2` look like similar lists of numbers, so one might hope that `obj[[1, 2]]` and `obj2[[1, 2]]` produce similar outputs. However, this does not occur, since the latter command will call the Index values associated to 1 and 2 in `obj2`, whereas the former command produces the first and second items in the Series `obj`.

```
obj2[[1, 2]]
```

produces the following output

```
1   -9
2    6
dtype: int64
```

while

```
obj[[1, 2]]
```

produces the output

```
c    7
a    5
dtype: int64
```

In fact, the last command will be deprecated in future version of Pandas.

Warning. Pandas allows slicing with labels, but the right endpoint is inclusive rather than exclusive.

```
obj.loc["c": "a"]
```

produces the following output

```
c    7
a    5
dtype: int64
```

Assignments for parts of the Series can be made in the following way.

```
obj.loc["c": "a"] = 0
obj
```

produces the following output

```
d    6
c    0
a    0
b   -9
dtype: int64
```

For DataFrames, it is again preferred to use the `loc` and `iloc` functions to query entries. Recall that

```
foods = {
    "apple": {"calories": 130, "sodium_mg": 2, "protein_g": .5},
    "pear": {"calories": 100, "sodium_mg": 2, "protein_g": .6}
}
frame = pd.DataFrame(foods)
```

outputs

```
            apple   pear
calories     130.0  100.0
sodium_mg   2.0     2.0
protein_g   0.5     0.6
```

Warning. Recall that accessing DataFrame entries has the opposite ordering of Numpy. That is `frame["apple"]["calories"]` outputs `130.0`, while the transposed command `frame.T["calories"]["apple"]` also outputs `130.0`. The latter ordering (row followed by column) matches Numpy's ordering for accessing array entries. Both `loc` and `iloc` follow the Numpy ordering.

```
frame.loc["calories"]
```

outputs

```
apple    130.0
pear     100.0
Name: calories, dtype: float64
```

In this case, `frame.iloc[0]` has the same output, i.e. a Series whose index is the columns of `frame`.

```
frame.loc[["protein_g", "calories"]]
```

outputs

```
            apple  pear
protein_g   0.5    0.6
calories   130.0   100.0
```

Multiple rows or columns can be selected in the following ways.

```
frame.loc[["protein_g", "calories"], "apple"]
```

outputs

```
protein_g      0.5
calories     130.0
Name: apple, dtype: float64
```

And

```
frame.iloc[[1, 2], [0, 1]]
```

outputs

```
           apple  pear
sodium_mg  2.0    2.0
protein_g  0.5    0.6
```

Series and DataFrames can be added. If either an Index or column entry does not appear in both summands, the corresponding output entry will be NaN. Built-in commands such as `add` have options such as `fill_value = 0`, which will replace the outputted NaN entries with 0.

Exercise 6.3. This exercise will use the following code.

```
data = {
    "state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
    "year": [2000, 2001, 2002, 2001, 2002, 2003],
    "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]
}
frame = pd.DataFrame(data)

populations = {
    "year": {0: 2000, 1: 2002, 3: 2004, 4: 2006},
    "pop": {0: 4, 2: 6, 3: 8, 4: 10}
}
frame2 = pd.DataFrame(populations)

ser = pd.Series([3, 6, 8, 9])

def f1(x):
    return x**2 +1
```

- Using the `add` function, add `frame` and `frame2` together (the syntax is `df.add(df2)`), and fill in any resulting NaN values to zeros.

- Apply the function `f1` to `frame`. (The syntax is `frame.map(f1)` .)
- For both NBA and WNBA players, answer the following question: Who has the highest 2pt + 3pt percentage (among those listed on both leaderboards) in a single season? (The percentage for a single player can be used across two different seasons.) To answer this question, you can find data from the following sites:
  WNBA Leaders
  NBA Leaders

6.4. **Case Study: MovieLens Database.** In this section we will study the Movie Lens 1M Small Dataset, available at

https://grouplens.org/datasets/movielens/

This dataset contains three separate files: `users.dat`, `ratings.dat` and `movies.dat`. The Readme file gives the following description of the dataset.

```
These files contain 1,000,209 anonymous ratings of approximately 3,900
movies made by 6,040 MovieLens users who joined MovieLens in 2000.

All ratings are contained in the file "ratings.dat" and are
in the following format:

UserID::MovieID::Rating::Timestamp

UserIDs range between 1 and 6040
MovieIDs range between 1 and 3952
Ratings are made on a 5-star scale (whole-star ratings only)
Timestamp is represented in seconds since the epoch as returned by time(2)
Each user has at least 20 ratings
USERS FILE DESCRIPTION

User information is in the file "users.dat" and is in the following format:

UserID::Gender::Age::Occupation::Zip-code

All demographic information is provided voluntarily by the users and is
not checked for accuracy. Only users who have provided some
demographic information are included in this data set.

Gender is denoted by a "M" for male and "F" for female

Age is chosen from the following ranges:

1: "Under 18"
18: "18-24"
25: "25-34"
35: "35-44"
45: "45-49"
50: "50-55"
```

```
56: "56+"
Occupation is chosen from the following choices:

0: "other" or not specified
1: "academic/educator"
2: "artist"
3: "clerical/admin"
4: "college/grad student"
5: "customer service"
6: "doctor/health care"
7: "executive/managerial"
8: "farmer"
9: "homemaker"
10: "K-12 student"
11: "lawyer"
12: "programmer"
13: "retired"
14: "sales/marketing"
15: "scientist"
16: "self-employed"
17: "technician/engineer"
18: "tradesman/craftsman"
19: "unemployed"
20: "writer"
MOVIES FILE DESCRIPTION

Movie information is in the file "movies.dat"
and is in the following format:

MovieID::Title::Genres

Titles are identical to titles provided by the
IMDB (including year of release)

Genres are pipe-separated and are selected from the following genres:

Action
Adventure
Animation
Children's
Comedy
Crime
Documentary
Drama
Fantasy
```

```
Film-Noir
Horror
Musical
Mystery
Romance
Sci-Fi
Thriller
War
Western
Some MovieIDs do not correspond to a movie due to accidental
duplicate entries and/or test entries

Movies are mostly entered by hand, so errors and inconsistencies may exist
```

We first load the three data files. Instead of our usual `utf-8` encoding, we switched to `latin-1` to avoid an exception.

```
unames = ["user_id", "gender", "age", "occupation", "zip"]
# users format:   UserID::Gender::Age::Occupation::Zip-code
users = pd.read_table(
    "users.dat",
    sep = "::",
    header = None,
    names = unames,
    engine = "python",
    encoding = "latin-1"
)

rnames = ["user_id", "movie_id", "rating", "timestamp"]
# ratings format:    UserID::MovieID::Rating::Timestamp
ratings = pd.read_table(
    "ratings.dat",
    sep="::",
    header = None,
    names = rnames,
    engine = "python",
    encoding = "latin-1"
)

mnames = ["movie_id", "title", "genres"]
# movies format:   MovieID::Title::Genres
movies = pd.read_table(
    "movies.dat",
    sep="::",
    header = None,
    names = mnames,
    engine = "python",
    encoding = "latin-1"
)
```

Then `users.head()` outputs

```
    user_id gender age occupation   zip
0   1       F       1  10           48067
1   2       M       56 16           70072
2   3       M       25 15           55117
3   4       M       45 7            02460
4   5       M       25 20           55455
```

and

```
ratings.head()
```

outputs

```
  user_id       movie_id    rating  timestamp
0 1             1193        5       978300760
1 1             661         3       978302109
2 1             914         3       978301968
3 1             3408        4       978300275
4 1             2355        5       978824291
```

and

```
movies.head()
```

outputs

```
  movie_id  title                                 genres
0 1         Toy Story (1995)                      Animation|Children's|Comedy
1 2         Jumanji (1995)                        Adventure|Children's|Fantasy
2 3         Grumpier Old Men (1995)               Comedy|Romance
3 4         Waiting to Exhale (1995)              Comedy|Drama
4 5         Father of the Bride Part II (1995)    Comedy
```

For convenience, we will merge everything into one table with the command

```
data = pd.merge(pd.merge(ratings, users), movies)
```

(Rearranging the order of `ratings,users,movies` would also rearrange the columns of `data`.) Pandas automatically merges the columns in the way you would want to merge them. (Note: This merge command works since we always merge two dataframes with at least one overlapping column name.)

```
user_id movie_id rating timestamp gender age occupation zip title genres
0  1 1193 5 978300760 F 1 10 48067 One Flew Over the Cuckoo's Nest (1975) Drama
1  2 1193 5 978298413 M 56 16 70072 One Flew Over the Cuckoo's Nest (1975) Drama
2 12 1193 4 978220179 M 25 12 32793 One Flew Over the Cuckoo's Nest (1975) Drama
3 15 1193 4 978199279 M 25 7 22903 One Flew Over the Cuckoo's Nest (1975) Drama
4 17 1193 5 978158471 M 50 1 95350 One Flew Over the Cuckoo's Nest (1975) Drama
5 18 1193 4 978156168 F 18 3 95825 One Flew Over the Cuckoo's Nest (1975) Drama
6 19 1193 5 982730936 M 1 10 48073 One Flew Over the Cuckoo's Nest (1975) Drama
7 24 1193 5 978136709 F 25 7 10023 One Flew Over the Cuckoo's Nest (1975) Drama
8 28 1193 3 978125194 F 25 1 14607 One Flew Over the Cuckoo's Nest (1975) Drama
```

```
9 33 1193 5 978557765 M 45 3 55421 One Flew Over the Cuckoo's Nest (1975) Drama
```

Also `data.iloc[0]` outputs

```
user_id                                               1
movie_id                                           1193
rating                                                5
timestamp                                     978300760
gender                                                F
age                                                   1
occupation                                           10
zip                                               48067
title         One Flew Over the Cuckoo's Nest (1975)
genres                                            Drama
Name: 0, dtype: object
```

Our first question is:

Question 6.4. What are the top rated movies, for those whose user specified gender is F or M?

To answer this question, we first compute the mean ratings of each title:

```
mean_ratings = data.pivot_table(
    "rating",
    index = "title",
    columns = "gender",
    aggfunc = "mean"
)
```

Here `mean_ratings.head()` outputs

```
gender                              F        M
title
$1,000,000 Duck (1971)          3.375000 2.761905
'Night Mother (1986)            3.388889 3.352941
'Til There Was You (1997)       2.675676 2.733333
'burbs, The (1989)              2.793478 2.962085
...And Justice for All (1979)   3.828571 3.689024
```

We can then sort `mean_ratings` according to the values in the `F` column. (Since we set `ascending` to be false, the list will be in descending order.)

```
top_f_ratings = mean_ratings.sort_values("F", ascending=False)
top_f_ratings.head()
```

which outputs

```
gender                                                  F           M
title
Clean Slate (Coup de Torchon) (1981)                    5.0         3.857143
Ballad of Narayama, The (Narayama Bushiko) (1958)       5.0         3.428571
Raw Deal (1948)                                         5.0         3.307692
Bittersweet Motel (2000)                                5.0         NaN
Skipped Parts (2000)                                    5.0         4.000000
```

Is it the case that these obscure movies are the most highly rated by the F users? That seems doubtful. What has happened is that a few F users rated these movies a 5.0. To eliminate this effect, let's filter out movie titles with less than 100 ratings.

```
ratings_by_title = data.groupby("title").size()
ratings_by_title.head()
```

which outputs

```
title
$1,000,000 Duck (1971)           37
'Night Mother (1986)             70
'Til There Was You (1997)        52
'burbs, The (1989)               303
...And Justice for All (1979)    199
dtype: int64
```

We then create a list of those titles with at least 100 ratings, and then input that into the `mean_ratings` DataFrame (whose index consists of movie titles)

```
active_titles = ratings_by_title.index[ratings_by_title >= 100]
mean_ratings_active = mean_ratings.loc[active_titles]
mean_ratings_active.head()
```

which outputs

```
gender                              F        M
title
'burbs, The (1989)                  2.793478 2.962085
...And Justice for All (1979)       3.828571 3.689024
10 Things I Hate About You (1999)   3.646552 3.311966
101 Dalmatians (1961)               3.791444 3.500000
101 Dalmatians (1996)               3.240000 2.911215
```

As before, we can then sort by F ratings.

```
top_f_ratings = mean_ratings_active.sort_values("F", ascending = False)
top_f_ratings.head()
```

which outputs

```
gender                                                F        M
title
Close Shave, A (1995)                                 4.644444 4.473795
Wrong Trousers, The (1993)                            4.588235 4.478261
General, The (1927)                                   4.575758 4.329480
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)         4.572650 4.464589
Wallace & Gromit: The Best of Aardman Animation (1996) 4.563107 4.385075
```

We can similarly sort the M user ratings:

```
top_m_ratings = mean_ratings_active.sort_values("M", ascending = False)
top_m_ratings.head()
```

which outputs

```
gender                                                 F          M
title
```

```
Godfather, The (1972)                                      4.314700 4.583333
Seven Samurai (The Magnificent Seven) (Shin...) (1954)  4.481132 4.576628
Shawshank Redemption, The (1994)                           4.539075 4.560625
Raiders of the Lost Ark (1981)                             4.332168 4.520597
Usual Suspects, The (1995)                                 4.513317 4.518248
```

What do we observe? With the top F ratings, it looks like a lot of Children's movies are highly rated. Why is that? Maybe a lot of child users are putting in reviews and skewing the ratings. To test that hypothesis, let's filter out the children users (i.e. those whose age is 1 in the DataFrame `data`.)

```
data_adults = data[data["age"] > 1]
ratings_by_title_adults = data_adults.groupby("title").size()
active_titles_adults = \
    ratings_by_title_adults.index[ratings_by_title_adults >= 100]
mean_ratings_active_adults = mean_ratings.loc[active_titles_adults]
top_f_adult_ratings = \
    mean_ratings_active_adults.sort_values("F", ascending = False)
top_f_adult_ratings.head()
```

which outputs

```
gender                                                  F        M
title
Close Shave, A (1995)                                   4.644444 4.473795
Wrong Trousers, The (1993)                              4.588235 4.478261
General, The (1927)                                     4.575758 4.329480
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)           4.572650 4.464589
Wallace & Gromit: The Best of Aardman Animation (1996) 4.563107 4.385075
```

With the same movies appearing as before even after we removed the child accounts, it seems we can reasonably conclude that the F accounts are watching movies with a child.

We now move on to a separate question.

Question 6.5. What movies have the largest disagreement by user reported gender?

To answer this question, let's first check for rating disagreement by gender

```
mean_ratings_active["diff"] = mean_ratings_active["F"] \
                             - mean_ratings_active["M"]
sorted_by_diff = mean_ratings_active.sort_values("diff")
sorted_by_diff.head()
```

which has output

```
gender                                          F        M         diff
title
Friday the 13th Part V: A New Beginning (1985) 1.272727 2.165049 -0.892321
Friday the 13th Part VI: Jason Lives (1986)    1.500000 2.291667 -0.791667
Lifeforce (1985)                                2.250000 2.994152 -0.744152
Marked for Death (1990)                         2.100000 2.837607 -0.737607
Quest for Fire (1981)                           2.578947 3.309677 -0.730730
```

and

```
sorted_by_diff.tail()
```

has output

```
gender                      F        M        diff
title
Home Alone 3 (1997)         2.486486 1.683761 0.802726
Air Bud (1997)              3.057143 2.233766 0.823377
Dirty Dancing (1987)        3.790378 2.959596 0.830782
Cutthroat Island (1995)     3.200000 2.341270 0.858730
Pet Sematary II (1992)      2.833333 1.858696 0.974638
```

It seems like some children's movies are appearing again, so let's filter out the child accounts like before

```
mean_ratings_active_adults["diff"] = mean_ratings_active_adults["F"] \
                                    - mean_ratings_active_adults["M"]
sorted_by_diff_adults = mean_ratings_active_adults.sort_values("diff")
sorted_by_diff_adults.head()
```

which has output

```
gender                                           F        M         diff
title
Friday the 13th Part V: A New Beginning (1985) 1.272727 2.165049 -0.892321
Friday the 13th Part VI: Jason Lives (1986)    1.500000 2.291667 -0.791667
Lifeforce (1985)                               2.250000 2.994152 -0.744152
Marked for Death (1990)                        2.100000 2.837607 -0.737607
Quest for Fire (1981)                          2.578947 3.309677 -0.730730
```

and

```
sorted_by_diff_adults[::-1].head()
```

has output

```
gender                                           F        M        diff
title
Pet Sematary II (1992)                           2.833333 1.858696 0.974638
Cutthroat Island (1995)                          3.200000 2.341270 0.858730
Dirty Dancing (1987)                             3.790378 2.959596 0.830782
Home Alone 3 (1997)                              2.486486 1.683761 0.802726
To Wong Foo, Thanks for Everything! ... (1995)  3.486842 2.795276 0.691567
```

The last table seemed a bit surprising. Perhaps we should also filter out movies with less than 50 F users, since it could be that a few F users who enjoy scary movies are skewing the results. Before doing that, let's look at movies with the smallest difference in F versus M ratings.

```
np.abs(sorted_by_diff_adults).sort_values("diff").head()
```

This outputs

```
gender                                       F        M        diff
title
Celebration, The (Festen) (1998)             4.307692 4.307692 0.000000
Fled (1996)                                  2.571429 2.571429 0.000000
```

```
Living Out Loud (1998)                     3.223529 3.223404 0.000125
Tender Mercies (1983)                      3.905405 3.905263 0.000142
Winnie the Pooh and the Blustery Day (1968) 3.986301 3.986486 0.000185
```

Now let us filter out movies with less than 50 F users.

```
ratings_by_title_f = \
    data_adults[data_adults["gender"] == "F"].groupby("title").size()
active_titles_f = ratings_by_title_f.index[ratings_by_title_f >= 50]
mean_ratings_trim_f = mean_ratings.loc[active_titles_f]
top_f_trim_ratings = \
    mean_ratings_trim_f.sort_values("F", ascending = False)
top_f_trim_ratings.head()
mean_ratings_trim_f["diff"] = mean_ratings_trim_f["F"] \
                            - mean_ratings_trim_f["M"]
sorted_by_diff_f = mean_ratings_trim_f.sort_values("diff")
sorted_by_diff_f.tail()
```

This outputs

```
gender                                          F        M        diff
title
Jane Eyre (1996)                                3.839286 3.192308 0.646978
Orlando (1993)                                  3.862745 3.190476 0.672269
Jumpin' Jack Flash (1986)                       3.254717 2.578358 0.676359
To Wong Foo, Thanks for Everything! ... (1995)  3.486842 2.795276 0.691567
Dirty Dancing (1987)                            3.790378 2.959596 0.830782
```

Here the scary movies were removed, confirming our suspicion that a few F users were rating them highly.

Question 6.6. We observed the maximum difference between F and M mean user ratings is about 1.

Can this observation be explained by randomness?

That is, if users are ranking movies in an entirely random way, would we make this same observation? The following simulation seems to suggest that random ratings could replicate this observation. However, this does not imply that viewers are making random ratings.

In the following simulation, we consider 100 F users who rate 4000 movies, and 100 M users who rate the same 4000 movies. We find the maximum difference of mean F versus mean M rating is around .7, which agrees with our observed maximum difference above.

```
f_ratings_sim = np.ceil(5 * np.random.rand(100, 4000))
m_ratings_sim = np.ceil(5 * np.random.rand(100, 4000))
f_mean_sim = np.mean(f_ratings_sim, axis = 0)
m_mean_sim = np.mean(m_ratings_sim, axis = 0)
diff_sim = f_mean_sim - m_mean_sim
np.max(diff_sim)
```

Question 6.7. Which movies are most divisive?

That is, which movies have the largest standard deviation in their ratings? To answer this question, we check the standard deviation of the ratings of each title.

```
rating_std_by_title = data.groupby("title")["rating"].std()
rating_std_by_title_active = rating_std_by_title.loc[active_titles]
```

We display the largest standard deviation

```
rating_std_by_title_active.sort_values(ascending = False)[:10]
```

```
title
Plan 9 from Outer Space (1958)                       1.455998
Beloved (1998)                                       1.372813
Godzilla 2000 (Gojira ni-sen mireniamu) (1999)       1.364700
Texas Chainsaw Massacre, The (1974)                  1.332448
Dumb & Dumber (1994)                                 1.321333
Crash (1996)                                         1.319636
Blair Witch Project, The (1999)                      1.316368
Natural Born Killers (1994)                          1.307198
Down to You (2000)                                   1.305310
Cemetery Man (Dellamorte Dellamore) (1994)           1.300647
Name: rating, dtype: float64
```

```
rating_std_by_title_active.sort_values(ascending = True)[:10]
```

```
title
Close Shave, A (1995)                            0.667143
Rear Window (1954)                               0.688946
Great Escape, The (1963)                         0.692585
Shawshank Redemption, The (1994)                 0.700443
Wrong Trousers, The (1993)                       0.708666
Central Station (Central do Brasil) (1998)       0.709393
Never Cry Wolf (1983)                            0.721782
Soldier's Story, A (1984)                        0.725206
Raiders of the Lost Ark (1981)                   0.725647
Seven Days in May (1964)                         0.729639
Name: rating, dtype: float64
```

Question 6.8. Can we predict the user specified gender just from their movie ratings?

Observe that

```
np.sum(users["gender"] == "M") / len(users)
```

outputs .717..., so the percentage of correct predictions should ideally be higher than this number.

Here is an attempt to answer Question 6.8. For each user, we check the titles they rate and compare the user rating to the average F rating of that title. We then do the same comparison for the average M rating. We then classify the user as M or F according to their deviation from the average M or F rating. Actually, since we compare these two averages, we are just comparing the mean F rating versus the mean M rating of each title the user has rated. That is, our algorithm only uses the titles rated, without taking into account the individual user ratings.

```
track_sum = 0
from tqdm import tqdm
```

```
for i in tqdm(range(len(users))):
    current_gender = users["gender"][i]
    current_id = i + 1
    current_data = data[data["user_id"] == current_id].set_index("title")
    #current_sorted = current_data.reindex(index = sorted_by_diff.index)
    current_nan = current_data.isna()
    current_data = current_data[~current_data.isna()]
    current_data.dropna(inplace = True)
    #current_data now has all ratings for user i, indexed by title
    f_score = (current_data["rating"] - mean_ratings["F"][current_data.index]) ** 2
    m_score = (current_data["rating"] - mean_ratings["M"][current_data.index]) ** 2
    f_avg = np.mean(f_score)
    m_avg = np.mean(m_score)
    if f_avg < m_avg:
        # then predict F
        gender_predict = "F"
    else:
        gender_predict = "M"

    if current_gender == gender_predict:
        # then prediction was correct
        track_sum += 1

print("Percentage prediction correct:", track_sum / len(users))
```

The output is 75.24...%.

Exercise 6.9. Try to modify the above code to predict more than 72.7% of user specified gender of the users from the Movie Lens 1M data. Alternatively, if you want, try to use some other classification algorithm we have discussed in order to get a better prediction percentage, just using the user ratings.

Question 6.10. Can we predict a user's movie rating for a movie they have not yet rated?

In this dataset, each user does not rate every movie. We would like to predict what movies a user will like, using their viewing data. Let A be the $m \times n$ matrix, where m is the number of users, n is the number of movie titles, and $A_{ij} \in \{1, 2, 3, 4, 5\}$ is equal to the rating of user i for movie title j. In this dataset, we have $m = 6040$ and $n = 3952$ and A has about $1,000,000$ known entries $E \subseteq \{(i, j)\colon 1 \leq i \leq 6040,\ 1 \leq j \leq 3952\}$. That is, the dataset contains about one million movie ratings which leaves more than 20 million entries of A unknown (unobserved).

```
rows = data["user_id"]
cols = data["movie_id"]
entries = data["rating"]
A = np.empty([6040, 3952], dtype = "uint8")
A[rows.values - 1, cols.values - 1] = entries.values
```

If we make no assumptions about A, then its entries can be filled in arbitrarily. However, there are only so many different types of people with different types of movie preferences.

That is, many of the rows of A should be identical or nearly identical. That is, we should be able to assume that the rank of A is low, e.g. less than 100. So, one way we can try to recover the unobserved entries of A is try to minimize the rank of a real $m \times n$ matrix B, subject to the constraint that $B_{ij} = A_{ij}$ for all $(i,j) \in E$. Equivalently, we could try to minimize the number of nonzero singular values of B, subject to the constraint that $B_{ij} = A_{ij}$ for all $(i,j) \in E$.

Unfortunately this problem is NP-hard [BK15]. So, instead of minimizing the number of nonzero singular values of B, we can try to minimize the sum of the singular values of B, subject to the constraint that $B_{ij} = A_{ij}$ for all $(i,j) \in E$. This problem is a convex optimization problem, but it can be fairly slow and does not parallelize, so it can be impractical for a matrix of large size (with 20 million entries).

So, an alternative approach is to fix an $r \geq 1$ and to find an $m \times r$ real matrix U and an $n \times r$ real matrix V that minimizes

$$\sum_{(i,j)\in E} (A_{ij} - (UV^T)_{ij})^2.$$

Since UV^T has rank at most r, the matrix UV^T will be a low rank approximation to the matrix A. Also, the problem is phrased in this way since, if U is fixed, then we can minimize over V, and if V is fixed we can minimizer over U. That is, we can alternate between solving two least squares minimization problems. To speed things up further, we can minimize over each row of U at a time, then over each row of V at a time, as in the following code.

```
from numpy.linalg import solve

def matrix_completion_als(A, mask, rank=5, iterations=100, reg_param=0.01):

    num_users, num_items = A.shape
    U = np.random.normal(scale=1./rank, size=(num_users, rank))
    V = np.random.normal(scale=1./rank, size=(num_items, rank))

    # Alternating Least Squares (ALS)
    for iteration in range(iterations):
        # Update U, keeping V fixed
        for i in range(num_users):
            mask_row = mask[i, :]
            V_masked = V[mask_row, :]
            A_masked = A[i, mask_row]
            if len(A_masked) > 0:
                U[i, :] = solve(
                    V_masked.T @ V_masked + reg_param * np.eye(rank),
                    V_masked.T @ A_masked
                )

        # Update V, keeping U fixed
        for j in range(num_items):
            mask_col = mask[:, j]
            U_masked = U[mask_col, :]
            A_masked = A[mask_col, j]
            if len(A_masked) > 0:
                V[j, :] = solve(
                    U_masked.T @ U_masked + reg_param * np.eye(rank),
                    U_masked.T @ A_masked
                )

        # Optionally, print out the loss to monitor convergence
        loss = np.sum((mask * (A - U @ V.T))**2)
        print(f"Iteration {iteration+1}/{iterations}, Loss: {loss}")

    # Return the completed matrix
    return U @ V.T
```

This function can be called as

```
completed_matrix = matrix_completion_als(
    A,
    A != 0,
    rank = 10,
    iterations = 50,
    reg_param = 0.1
)
```

However, one downside of this approach is that $(UV^T)_{ij}$ might not be equal to A_{ij} when $(i, j) \in E$. Still, at least we have some prediction for the missing movie ratings. For example

```
A[:5, :5]
```

has output

```
array([[5, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=uint8)
```

which consists mostly of empty ratings. But

```
completed_matrix[:5, :5]
```

has output

```
array([[4.17517743, 3.50608825, 4.40508423, 3.40020978, 4.69459213],
       [4.47761245, 3.35353216, 3.04344264, 3.3516488 , 3.5498769 ],
       [3.44207249, 3.2142867 , 2.95733293, 2.90528044, 2.57136808],
       [4.28084915, 2.02099364, 1.17336766, 2.13082553, 1.84848784],
       [3.36399502, 2.06767066, 1.6148004 , 1.51961442, 1.17258333]])
```

Observe that the output UV^T has entries that do not agree with A, non-integer entries (and it can even have negative entries). To see why it is unlikely for UV^T

7. Web APIs and Data Cleaning

7.1. **Zillow Sales Data.** In this section, we will use the `requests` package to explore some datasets from the internet. You can install this package from the command line with the command

```
pip install requests
```

Here is an example of how to use the requests package.

```
import requests
url = "https://api.github.com/repos/pandas-dev/pandas/issues"
resp = requests.get(url)
# check for HTML errors
resp.raise_for_status()
# should output <Response [200]> to indicate success
resp
# convert to a list of dictionaries
data = resp.json()
```

For example, `data[0]` will be a dictionary object, with abbreviated form

```
{'url': 'https://api.github.com/repos/pandas-dev/pandas/issues/59291',
 'repository_url': 'https://api.github.com/repos/pandas-dev/pandas',
 ...
'id': 2420993845,
 'node_id': 'PR_kwDOAA0YD851_iH2',
 'number': 59291,
 'title': 'DOC: Fix sentence fragment in string methods',
 ...
 'performed_via_github_app': None,
 'state_reason': None}
```

If we just wanted to focus on a few of the items in the dictionary, we could pass it to a DataFrame with e.g.

```
 issues = pd.DataFrame(data, columns=["number", "title","labels", "state"])
```

For another example, I would like to analyze some zillow sales data. This data is available under "sales" at the website

```
https://www.zillow.com/research/data/
```

After downloading the file, we can pass the .csv file to a dataframe with

```
data = pd.read_csv('Metro_sales_count_now_uc_sfrcondo_month.csv')
```

Here is an abbreviated example of this data, which gives the number of home sales during each month in different metro areas in the USA (though the first row is just all sales in the USA).

```
  RegionID SizeRank RegionName RegionType StateName 2008-02-29 2008-03-31...
0 102001   0        United States country NaN     204850.0    237683.0  ...
1 394913   1        New York, NY     msa  NY        8585.0      8965.0  ...
2 753899   2        Los Angeles, CA  msa  CA        4159.0      5052.0  ...
3 394463   3        Chicago, IL      msa  IL        5882.0      7411.0  ...
4 394514   4        Dallas, TX       msa  TX        5001.0      5664.0  ...
... ... ...
```

I would like to just examine some trends relating the different metro areas. I will first remove the first row and first few columns, then delete any columns with NaN values, and convert to a Numpy array.

```
sales_cut = data.iloc[1:, 5:]
sales = sales_cut.dropna(axis = 1).values
```

I am curious if we can classify these metro areas into similar classes. Since some metro areas are naturally larger than others, before applying k-means clustering, I will normalize each row of the `sales` array so they are all vectors of the same length.

```
import matplotlib.pyplot as plt
row_length = np.linalg.norm(sales, axis = 1)
sales_normed = (sales.T/ row_length).T
nrow, ncol = sales_normed.shape
U, D_vector, V = np.linalg.svd(sales_normed)
# number of principal components
q = 2
D_truncated = np.zeros([nrow, q])
D_truncated[:q, :q] = np.diag(D_vector[:q])
pca_data = U @ D_truncated
fig, ax = plt.subplots()
ax.scatter(pca_data[:, 0], pca_data[:, 1])
for i in np.arange(nrow):
    ax.annotate(sales_values[i+1,2],
      xy = (pca_data[i, 0]+.002, pca_data[i, 1]), size = 4)
ax.set_xlabel('1st component')
ax.set_ylabel('2nd component')
plt.savefig('zillow_sales.pdf')
plt.show()
```

For some reason, Little Rock has much different home sale behavior than many other places. To try to find an explanation, I will repeat the above PCA plot for the differences in number of sales in consecutive months, rather than the number of sales themselves.

```
sales_change = sales[:, 0:-1] - sales[:, 1:np.shape(sales)[1]]
```

Repeating the same code for `sales_change` in place of `sales` we get the plot below.

Once again Little Rock is a bit of an outlier, but I could not find a good explanation for why its behavior is different than other metropolitan areas. To try to figure out what is going on, let's check for correlations in the prices with each other. Since Little Rock is an outlier, we expect that it will have a relatively lower correlation with other metropolitan areas.

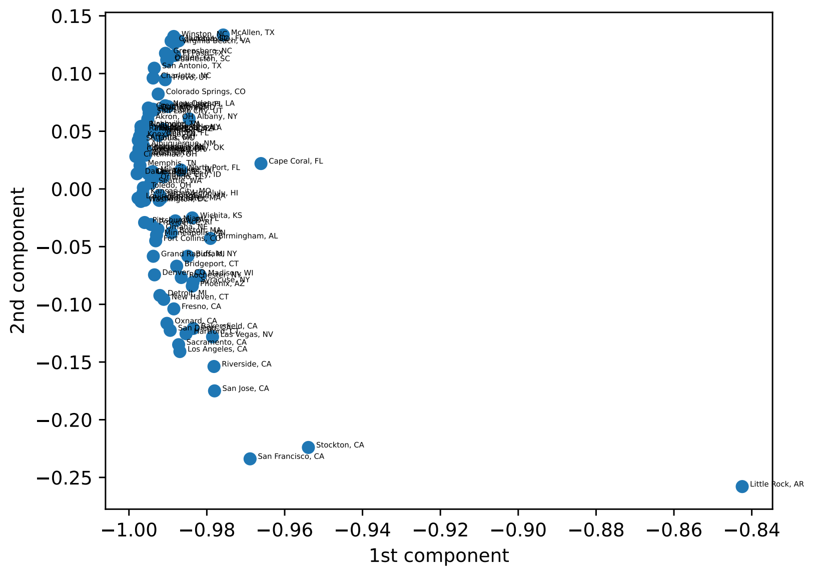


FIGURE 4. Plot of PCA output in two dimensions. Zillow Monthly Sales Data, 2008–2024, Normalized

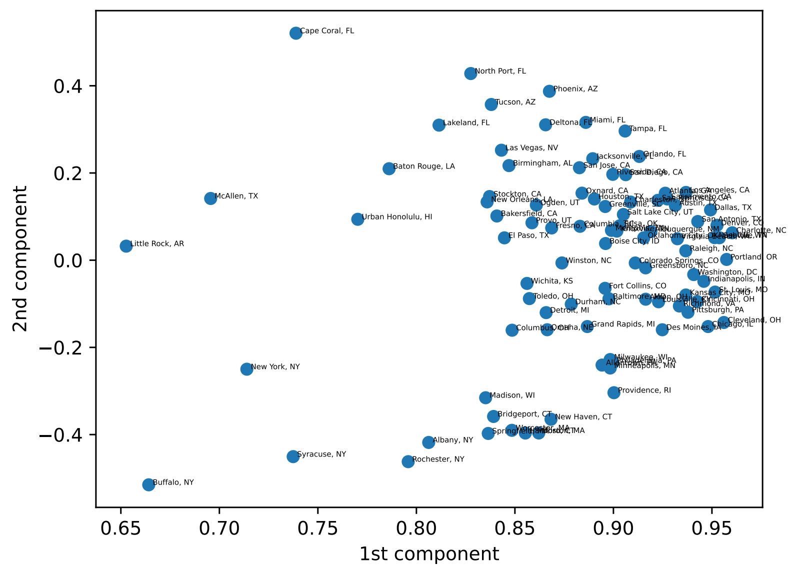


FIGURE 5. Plot of PCA output in two dimensions. Zillow Monthly Sales Data, 2008–2024, Changes between months, Normalized

```
data_trim = pd.DataFrame(
    {data.loc[i]["RegionName"] : data.loc[i]["2008-02-29":]
     for i in range(94)}
)
corr_df = data_trim.corr()
print(corr_df)
```

When we print the correlations, and then sum up each column, we see there are two cities with particular low column sums. Most column sums are around 60 or 70, but two of them are quite small, i.e. less than 30.

```
print(print(np.sum(corr_df.to_numpy(), axis=0)))
print(z[np.sum(corr_df, axis = 0) < 30])
```

These cities are Stockton, CA and Little Rock, AR. So, it seems that the number of housing sales of Little Rock, AR is typically uncorrelated with other metropolitan areas. To see if there is any regional explanation, let's check the correlation of California metropolican areas.

```
ca_index = ["CA" in x for x in corr_df.index]
corr_df[ca_index].T[ca_index]
```

outputs

```
Los Angeles, CA San Francisco, CA Riverside, CA San Diego, CA Sacramento, CA
    San Jose, CA Fresno, CA Bakersfield, CA Oxnard, CA Stockton, CA
Los Angeles, CA   1.000 0.877 0.876 0.967 0.941 0.915 0.894 0.854 0.954 0.601
San Francisco, CA 0.877 1.000 0.817 0.833 0.859 0.933 0.724 0.740 0.822 0.732
Riverside, CA     0.876 0.817 1.000 0.862 0.870 0.836 0.849 0.906 0.848 0.823
San Diego, CA     0.967 0.833 0.862 1.000 0.950 0.889 0.897 0.850 0.955 0.577
Sacramento, CA    0.941 0.859 0.870 0.950 1.000 0.884 0.898 0.867 0.950 0.677
San Jose, CA      0.915 0.933 0.836 0.889 0.884 1.000 0.799 0.808 0.859 0.644
Fresno, CA        0.894 0.724 0.849 0.897 0.898 0.799 1.000 0.887 0.887 0.578
Bakersfield, CA   0.854 0.740 0.906 0.850 0.867 0.808 0.887 1.000 0.859 0.676
Oxnard, CA        0.954 0.822 0.848 0.955 0.950 0.859 0.887 0.859 1.000 0.596
Stockton, CA      0.601 0.732 0.823 0.577 0.677 0.644 0.578 0.676 0.596 1.000
```

It seems that Stockton, CA has a slightly lower correlation to other California metropolitan areas, though the lack of regional effect is more pronounced for Little Rock, as we see below.

```
ar_index = ["AR" in x or "OK" in x or "MO" in x or "LA" in x
            for x in corr_df.index]
corr_df[ar_index].T[ar_index]
```

outputs

```
St. Louis, MO Kansas City, MO Oklahoma City, OK New Orleans, LA
    Tulsa, OK Baton Rouge, LA Little Rock, AR
St. Louis, MO       1.000 0.951 0.947 0.893 0.960 0.918 0.222
Kansas City, MO     0.951 1.000 0.881 0.798 0.906 0.884 0.275
Oklahoma City, OK   0.947 0.881 1.000 0.906 0.954 0.885 0.302
New Orleans, LA     0.893 0.798 0.906 1.000 0.897 0.906 0.258
Tulsa, OK           0.960 0.906 0.954 0.897 1.000 0.910 0.228
Baton Rouge, LA     0.918 0.884 0.885 0.906 0.910 1.000 0.199
Little Rock, AR     0.222 0.275 0.302 0.258 0.228 0.199 1.000
```

As we see, Little Rock has a distinctly lower correlation with its regional neighbors.

I was just curious how the other correlation numbers look, so I sorted then with e.g.

```
print(np.sort(corr_df["Stockton, CA"]))
```

A few of these entries actually have negative correlation. To see which cities, we use

```
neg_corr = [x for i,x in enumerate(corr_df.index)
                if corr_df["Stockton, CA"][x]<0]
print(neg_corr)
```

whose output is

```
['Jacksonville, FL', 'Birmingham, AL', 'McAllen, TX', 'El Paso, TX',
   'Little Rock, AR', 'Lakeland, FL', 'Deltona, FL']
```

This observation agrees with conventional wisdom that there is currently a net flow from California to Texas and Florida.

It turns out the most correlated with Stockton, CA is Riverside, CA with a .823 correlation, and the most correlated with Little Rock, AR is Birmingham, AL with a .671 correlation.

7.2. **Google Finance Data.** In the following example, we are using Google Finance to find some current stock prices for some technology stocks. The `requests` package is used to query the domain
`https://www.google.com/finance` for the stock data, for each of four stocks (Meta, Microsoft, Google, and Nvidia). After reading the text output of the URL query, we find that we can extract the price data whenever the current year (2024) appears, followed by at most 100 other characters, and then finally the string `2,2,4`. For example, here is part of the web page output:

```
[[2024,7,3,9,34,null,null,[-14400]],[508.02,-1.4800000000000182,
    -0.002904808635917602,2,2,4],20991],
```

The stock price is 508.02, the date and time for this price is July 3, 2024 at 9:34. We will not use the other parts of this string.

In order to find all of the prices that appear, we use the command

```
extracted = re.findall(r"\[2024([a-z0-9,.\-\[\]]{0,100}),2,2,4\]", data)
```

Inside the `findall` command, the letter `r` denotes a raw string, and then the string search command follows. We first look for the string `[2024` (since the bracket is a special character, we need to add a slash to search for it in the `findall` function). After finding `[2024`, we then nest inside parentheses the part of the string we want to output. Inside these parentheses, we have the command `[a-z0-9,.\-\[\]]{0,100}`. The `[a-z0-9,.\-\[\]]` means we are looking for any alphanumeric lower case character, or any of the characters `,.-[]` (In the latter two cases we again have to add a slash to search for those special characters). Then, the `{0,100}` command denotes we can look for at most 100 of those characters specified by the `[a-z0-9,.\-\[\]]` command. Finally, the string `,2,2,4` is the end of the part where the stock price data appears.

Part of the `extracted` list of strings is printed below:

```
',7,3,9,34,null,null,[-14400]],[508.02,-1.4800000000000182,
    -0.002904808635917602'
```

Since we only care about the price (508.02) and the day/time of the stock price (July 3 at 9:34), we will run through each string element of the list `extracted` to delete the other parts of this string, and also split it into two different lists (one for the price, another for the time). This deletion procedure uses the functions `clean_string` and `clean date`.

Finally, for some reason the different stocks have different numbers of prices. So, for convenience, I just throw out all stock price entries that are longer than the shortest stock price list.

```
import requests
import re

stocks = ["META", "MSFT", "GOOG", "NVDA"]
total_data = []
data_len = []

def clean_string(s):
    if "[" in s:
        index = s.find("[")
        s=s[index+1:]
    if "," in s:
        index = s.find(",")
        return s[0:index]
    return s

def clean_date(s):
    if "n" in s:
        index = s.find("n")
        s=s[:index]
        s=s+"0"
    return s

for j in range(len(stocks)):
    url = "https://www.google.com/finance/quote/" + stocks[j] + ":NASDAQ?hl=en"
    resp = requests.get(url)
    data = resp.text
    #print(data)
    extracted = re.findall(r"\[2024([a-z0-9,.\-\[\]]{0,100}),2,2,4\]", data)
    time_stamps = []
    stock_prices = []
    for i in range(len(extracted)):
        if "." in extracted[i] and len(extracted[i])>5:
            time_stamps.append(clean_date(extracted[i][1:11]))
            stock_prices.append(clean_string(extracted[i][31:38]))

    total_data.append(time_stamps)
    total_data.append(stock_prices)
    data_len.append(len(time_stamps))

print(total_data)
```

We could then plot all four stock prices in one plot as follows.

```
import matplotlib.pyplot as plt

# convert date string such as "7,19,9,32" to 7 + (19-1)/31 + (9-1)/(24*31) + (32-1)/(24*
def process_date(x):
    if x[-2] == ",":
        x = x[:-1] + "0" + x[-1]
    comma_index = [i for i, char in enumerate(x) if char == ","]

    for j in range(len(comma_index) - 1):
        if comma_index[j+1] - comma_index[j] == 2:
            # then add an extra 0 after comma_index[j]
            x = x[:(1+comma_index[j])] + "0" + x[(comma_index[j+1] - 1):]
    # so far, we replaced 7,19,9,32 with 7,19,09,32
    comma_index = [i for i, char in enumerate(x) if char == ","]
    integer_part = x[:comma_index[0]]
    #integer_part = integer_part.replace(",","")

    day = x[1 + comma_index[0]:comma_index[1]]
    hour = x[(1 + comma_index[1]):comma_index[2]]
    minute = x[(1 + comma_index[2]):]
    minute = minute.replace(",","")

    #divide day by number of days in that month
    if day in ['4', '6', '9', '11']:
        num_days = 30
    elif day == '2':
        num_days = 28
    else:
        num_days = 31

    day = (int(day) - 1) / num_days
    hour = (int(hour) - 1) / (24 * num_days)
    minute = (int(minute) - 1) / (60 * 24 * num_days)

    decimal_part = day + hour + minute

    return float(integer_part) + decimal_part

def process_date_list(x):
    out=[]
    for i in range(len(x)):
        out.append(process_date(x[i]))

    return out

fig, ax = plt.subplots()
colors = ["red", "blue", "green", "cyan"]
for k in range(len(stocks)):
    ax.scatter(
        process_date_list(total_data[2*k - 2]),
        [float(i) for i in total_data[2*k - 1]],
        c = colors[k],
        s=1
    )
ax.set_xlabel('time (m as integer, d/h/s as fraction)')
ax.set_ylabel('price (dollars per share)')
ax.tick_params(reset = True)
ax.legend(stocks)
plt.savefig('stock_prices.pdf')
plt.show()
```

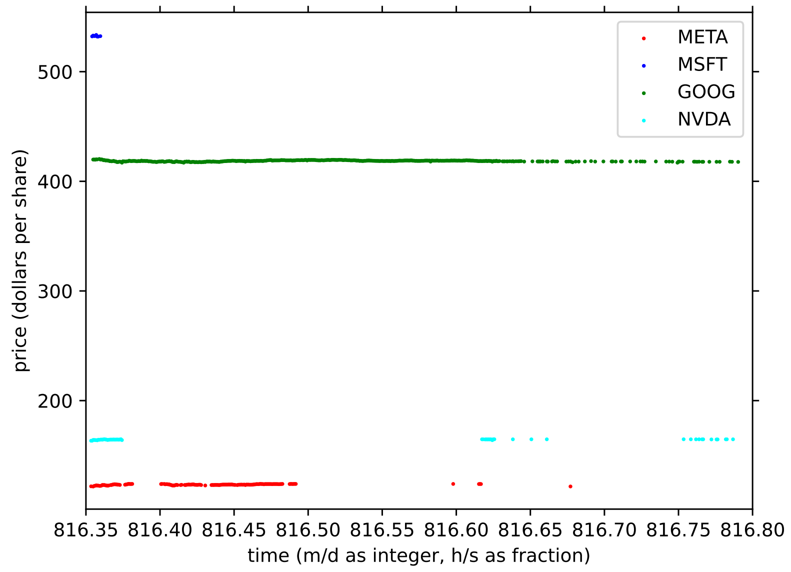


Figure 6. Some stock prices from Google Finance

It seems like we are missing some data though.

Exercise 7.1. Modify the above code to fill in the data values that appear to be missing from the plot.

Let's pass the stock data to a DataFrame and calculate the correlation of each stock price with each other one using the `corr()` method.

```
df = pd.DataFrame(
    {stocks[i] : dict(zip(total_data[2*i -2],
                          total_data[2*i -1])) for i in range(4)}
)
print(df)
print(df.corr())
```

```
             META    MSFT    GOOG    NVDA
8,16,9,30,  121.82     NaN     NaN   163.4
8,16,9,32,  121.64   532.9  419.83  164.11
8,16,9,33,  122.16  532.48  419.90  164.12
8,16,9,34,  122.36   532.6  419.73  163.98
8,16,9,35,  122.68  533.49  420.05  163.96
...            ...     ...     ...     ...
8,16,19,33     NaN     NaN     NaN   164.6
8,16,19,39     NaN     NaN     NaN   164.6
```

```
8,16,19,47     NaN     NaN     NaN   164.7
8,16,19,48     NaN     NaN     NaN   164.7
8,16,19,54     NaN     NaN     NaN   164.7

[479 rows x 4 columns]
          META      MSFT      GOOG      NVDA
META  1.000000 -0.060123  0.013079  0.623733
MSFT -0.060123  1.000000  0.288277  0.092158
GOOG  0.013079  0.288277  1.000000 -0.406910
NVDA  0.623733  0.092158 -0.406910  1.000000
```

Due to an oddity in the data, the nonempty values of the stock prices have almost empty overlap, resulting in some near zero correlations.

7.3. **eBay Sales Price Data.** In the following example, we are using eBay to find some recent sales of a USC Trojans Jersey. The `requests` package is used to query the domain `https://www.ebay.com` for the sales data. After reading the text output of the URL query, we find that we can extract the price data from three different prefixes, according to how the sales price is formatted on the screen (which denotes what type of auction occurred). These prefixes are: `class=POSITIVE>`, `POSITIVE ITALIC">`, and `STRIKETHROUGH POSITIVE">`. In each of these cases, a dollar sign occurs and then the sales price is listed, with two decimal places. For example, here is part of the web page output:

```
<span class=POSITIVE>$24.80</span>
```

The sales price here is 24.80. The sold data occurs in a different string, with the prefix `POSITIVE"><span>`, as follows:

```
signal POSITIVE"><span>Sold  Jul 4, 2024</span>
```

In order to find all of the prices that appear, we search for the three possible prefixes we listed above. Our search string put into the `re.findall` command is the following:

```
search_string = (
    r'class=POSITIVE>([\w$.,</]{10})'
    r'|POSITIVE ITALIC">([\w$.,</]{10})'
    r'|STRIKETHROUGH POSITIVE">([\w$.,</]{10})'
)
```

The outer parenthesis here enable a multiline string command (we need to use three `r` characters to concatenate a single raw string over multiple lines.) The two vertical bars correspond to logical "or" commands, so that we search for three different things. For the first string prefix, the command inside parentheses is `[\w$.,</]{10}`. This command asks for ten alphanumeric characters (`\w` represents any letter or number) or characters among `$,.</]` . As we can see from the above example, these characters do appear after the stated prefix. All instances of the `search_string` are then found via the following command:

```
    extracted_prices = re.findall(search_string, data)
```

Similarly, the sale date is found via the command

```
    extracted_dates = re.findall(r'POSITIVE"><span>Sold([\w .</]{7})', data)
```

Part of the `extracted_prices` list of tuples of strings is printed below:

```
('', '', '$900.00</s')
```

Since we ask for three different string matches, the output of `re.findall` is a 3-tuple of strings. The function `clean_price` removes the extraneous characters from this 3-tuple. We also delete the empty parts of the tuple.

Part of the `extraced_dates` list of strings is printed below

```
'  Jul 4'
```

The function `clean_date` removes the extra spaces from this string.

```
import requests
import re
import numpy as np

def clean_price(s):
    index1 = s.find("$")
    index2 = s.find(".")
    return s[index1+1:index2+3]

def clean_date(s):
    return s[2:]


url = """https://www.ebay.com/sch/i.html?_nkw=usc+trojans+jersey
        &_sacat=0&LH_Complete=1&LH_Sold=1"""
resp = requests.get(url)
data = resp.text
search_string=(
    r'class=POSITIVE>([\w$.,</]{10})'
    r'|POSITIVE ITALIC">([\w$.,</]{10})'
    r'|STRIKETHROUGH POSITIVE">([\w$.,</]{10})'
              )
extracted_prices = re.findall(search_string, data)
extracted_dates = re.findall(r'POSITIVE"><span>Sold([\w .</]{7})', data)

if len(extracted_prices) != len(extracted_dates):
    print("Error: Prices and dates have different lengths!  Fixing...")
    min_length = np.min([len(extracted_prices), len(extracted_dates)])
    extracted_prices = extracted_prices[:min_length]
    extracted_dates = extracted_dates[:min_length]

time_stamps = []
stock_prices = []
total_data=[]
for i in range(len(extracted_prices)):
    time_stamps.append(clean_date(extracted_dates[i]))
    for j in range(3):
        if len(extracted_prices[i][j])>0:
            price_index = j
    stock_prices.append(clean_price(extracted_prices[i][price_index]))

total_data.append(time_stamps)
total_data.append(stock_prices)

print(total_data)
```

8. Regression

In statistics and machine learning, we often use data to make predictions. In Section 4, we classified data into a discrete set of categories. For example, we classified flower measurements into three different species, and we classified newsgroups postings into six different categories in Exercise 4.14. These examples are called **classification** tasks, since the labels we give to datapoints come from a discrete set. In contrast, **regression** tasks assign datapoints a label from a continuous set. For example, we might predict someone's adult height given their height at age two.

So far, we have focused mostly on classification tasks. In the next Section 9, we will improve on an approach to handwriting classification from Exercise 4.17.

In this section, we will provide a few examples of regression.

8.1. **Linear Regression.** We have covered linear regression and least squares estimation in previous classes, so let's start with an example. We are going to plot the average price of electricity per kilowatt-hour in Los Angeles-Long Beach-Anaheim, CA, as provided at the following website, where we can download a `.csv` file.

https://fred.stlouisfed.org/series/APUS49A72610

We will use the `LinearRegression` function to find the best fit line to this data.

```
import numpy as np
import pandas as pd
import time as time
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

df = pd.read_csv("electric.csv", encoding="utf-8")
df = df.rename(columns={"DATE": "date", "APUS49A72610": "rate"})
# clean a missing entry
df[df.values[:, 1] == '.'] = ['1985-08-01', '0.083']

# convert date to an integer value
for i in range(len(df)):
    year = df.iloc[i, 0][0:4]
    month = df.iloc[i, 0][5:7]
    month_decimal = (int(month) - 1) / 12
    month_decimal = ((month_decimal * 100) // 1) / 100
    month_str = str(month_decimal)
    df.iloc[i, 0] = year + month_str[1:]

# reshape is required for one-dimensional data
regr = LinearRegression()
regr.fit(
    df.values[:, 0].astype(float).reshape(-1, 1),
    df.values[:, 1].astype(float)
)

print(regr.coef_[0], regr.intercept_)

print("Slope: ", regr.coef_[0], "Intercept: ", regr.intercept_)

# Plot outputs
plt.plot(
    df.values[:, 0].astype(float),
    df.values[:, 1].astype(float),
    color="red"
)
plt.plot(
    df.values[:, 0].astype(float),
    regr.coef_ * df.values[:, 0].astype(float) + regr.intercept_,
    color="blue", linewidth=3
)
plt.xlabel('Date, as an integer')
plt.ylabel('Electric rate (dollars per kilowatt hour)')
plt.savefig('electric.pdf')

plt.show()
```

```
Slope:  0.004084495925694919 Intercept:  -8.026650343894014
```

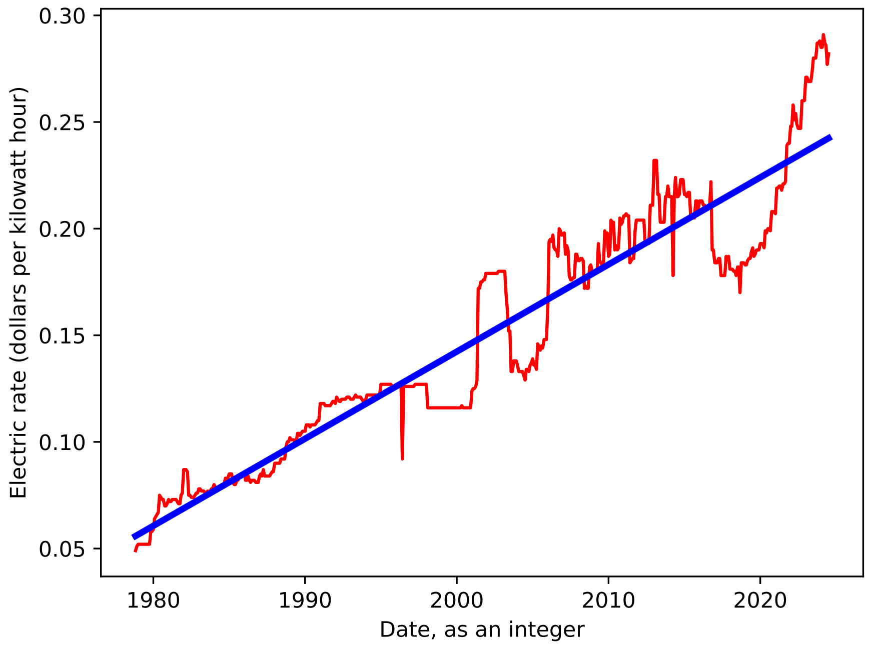


Figure 7. Average Price of Electricity per Kilowatt-Hour in Los Angeles-Long Beach-Anaheim, CA (CBSA), together with best linear fit

Since we found a slope of about 4×10^{-3}, this means the electric rate is going up at by approximately 4 cents every ten years.

Exercise 8.1. The electricity prices we analyzed behaved a bit differently over the last 7 years. Perform a linear regression just on data from 2017 to the present. What slope do you get?

Our electricity prices did not reflect inflation. Perform the same linear regression analysis (for the full dataset, and for the past 7 years) by adjusting for inflation. (This part of the exercise is intentionally open-ended. To get some usable inflation numbers, see e.g. https://fred.stlouisfed.org/series/FPCPITOTLZGUSA . For example, if electricity rates increased by three percent in one year when inflation is three percent for that whole year, then the inflation adjusted electricity rate would be constant over that year.)

8.1.1. *Multiple Linear Regression.*

```
import pandas as pd
import numpy as np
from sklearn.datasets import fetch_california_housing
import matplotlib.pyplot as plt
from pandas.plotting import scatter_matrix
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

housing = fetch_california_housing(as_frame=True)
housing = housing.frame
housing.head()
```

```
 MedInc HouseAge AveRooms AveBedrms Population AveOccup Latitude Longitude MedHouseVal
0 8.3252  41.0  6.984127  1.023810  322.0      2.555556 37.88   -122.23    4.526
1 8.3014  21.0  6.238137  0.971880  2401.0     2.109842 37.86   -122.22    3.585
2 7.2574  52.0  8.288136  1.073446  496.0      2.802260 37.85   -122.24    3.521
3 5.6431  52.0  5.817352  1.073059  558.0      2.547945 37.85   -122.25    3.413
4 3.8462  52.0  6.281853  1.081081  565.0      2.181467 37.85   -122.25    3.422
```

```
california_housing = fetch_california_housing(as_frame=True)
print(california_housing.DESCR)
```

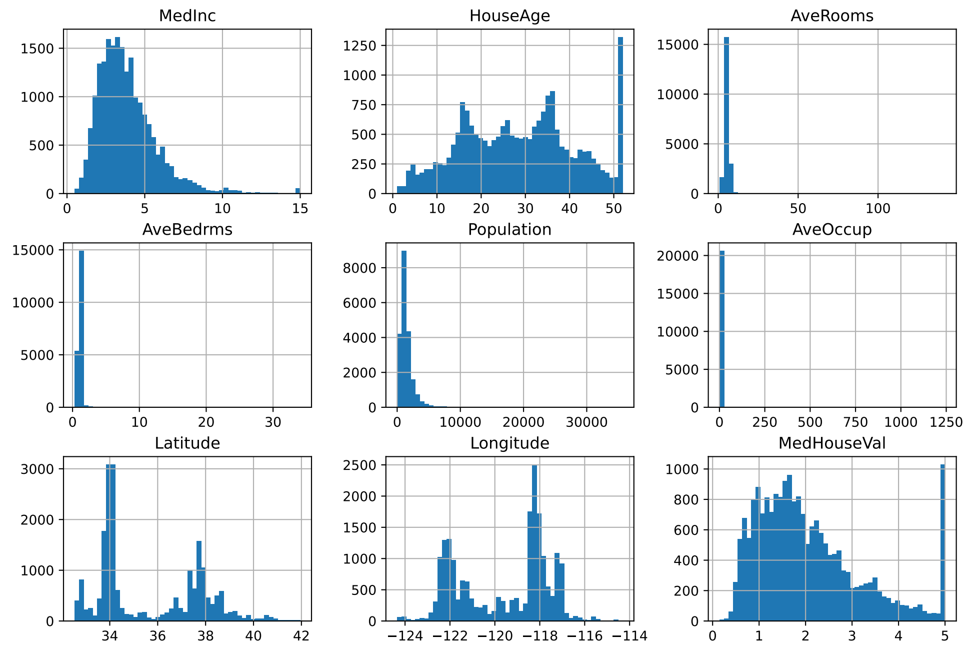
```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

:Number of Instances: 20640

:Number of Attributes: 8 numeric, predictive attributes and the target

:Attribute Information:
    - MedInc        median income in block group
    - HouseAge      median house age in block group
    - AveRooms      average number of rooms per household
    - AveBedrms     average number of bedrooms per household
    - Population    block group population
    - AveOccup      average number of household members
    - Latitude      block group latitude
    - Longitude     block group longitude

:Missing Attribute Values: None
```

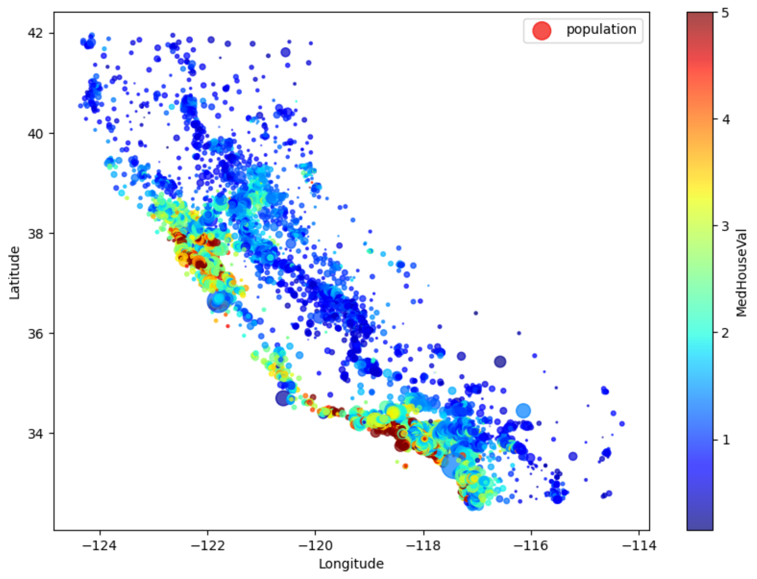
```
This dataset was obtained from the StatLib repository.
https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

The target variable is the median house value for California districts,
expressed in hundreds of thousands of dollars ($100,000).

This dataset was derived from the 1990 U.S. census, using one row per census
block group. A block group is the smallest geographical unit for which the U.S.
Census Bureau publishes sample data (a block group typically has a population
of 600 to 3,000 people).

A household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, these
columns may take surprisingly large values for block groups with few households
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. rubric:: References

- Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
  Statistics and Probability Letters, 33 (1997) 291-297
```

```
housing.hist(bins=50, figsize=(12,8))
plt.savefig('cahousing.pdf')
plt.show()
```

```
housing.plot(
    kind = "scatter",
    x = "Longitude",
    y = "Latitude",
    c = "MedHouseVal",
    cmap = "jet",
    colorbar = True,
    legend = True,
    sharex = False,
    figsize = (10,7),
    s = housing['Population']/100,
    label = "population",
    alpha = 0.7
)
plt.savefig('cahousing2.png')
plt.show()
```

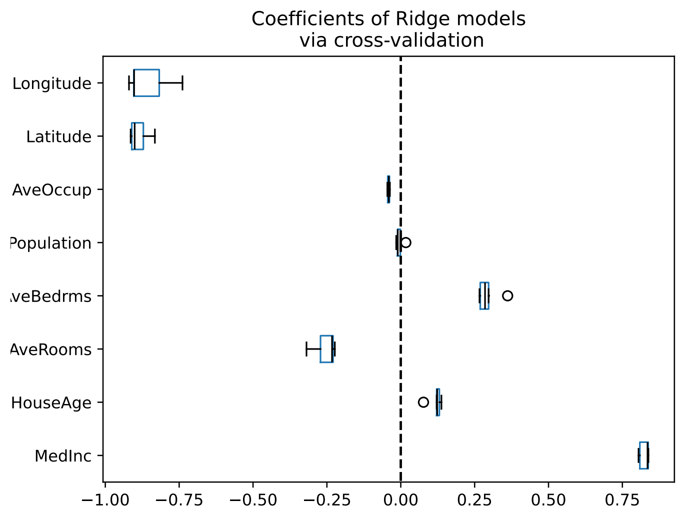
```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import RidgeCV
```





```
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_validate

alphas = np.logspace(-3, 1, num=30)
model = make_pipeline(StandardScaler(), RidgeCV(alphas=alphas))
```

```
cv_results = cross_validate(
    model,
    california_housing.data,
    california_housing.target,
    return_estimator=True,
    n_jobs=2,
)

coefs = pd.DataFrame(
    [est[-1].coef_ for est in cv_results["estimator"]],
    columns=california_housing.feature_names,
)
color = {"whiskers": "black", "medians": "black", "caps": "black"}
coefs.plot.box(vert=False, color=color)
plt.axvline(x=0, ymin=-1, ymax=1, color="black", linestyle="--")
_ = plt.title("Coefficients of Ridge models\n via cross-validation")
plt.savefig('cahousing3.pdf')
```



As expected, the largest coefficients occur for longitude, latitude and median income, i.e. these are the most significant factors for determining the price of a house.

8.2. **Logistic Regression.** In our next example, we will use logistic regression to classify emails as spam or not spam (ham). We will first introduce logistic regression.

Denote the **logistic function** as

$$h(x) := \frac{1}{1+e^{-x}}, \qquad \forall\, x \in \mathbb{R}.$$

Note that $\lim_{x\to\infty} h(x) = 1$ and $\lim_{x\to-\infty} h(x) = 0$.

Let $X_1, \ldots, X_m$ be i.i.d. real-valued random variables. Let $g\colon \mathbb{R} \to \{0, 1\}$ be an unknown function, and let $Y_i := g(X_i)$ for all $1 \leq i \leq m$. For example, $X_1, \ldots, X_m$ could be the blood pressures of m people, and $g(X_i) = 1$ if person $i \in \{1, \ldots, m\}$ has had a heart attack, whereas $g(X_i) = 0$ if person i has not had a heart attack. In this way, g classifies the data has having or not having a certain trait. For another example, X_i could be some characteristic of the i^{th} received email, $g(X_i) = 1$ if email $i \in \{1, \ldots, m\}$ is spam, whereas $g(X_i) = 0$ if email i is not spam.

By our assumptions, $Y_1, \ldots, Y_m$ are i.i.d. Bernoulli random variables with some unknown probability $0 \leq p \leq 1$ such that $p = \mathbf{P}(Y_1 = 1)$. Since the logistic function smoothly transitions from value 0 to value 1, we make the heuristic assumption that there are some unknown parameters $a, b \in \mathbb{R}$ such that

$$p \approx h(ax + b) \approx g(x).$$

The likelihood function is then

$$\ell(a, b) := \prod_{i=1}^{m} p^{y_i}(1-p)^{1-y_i} = \prod_{i=1}^{m} [h(ax_i + b)]^{y_i}[1 - h(ax_i + b)]^{1-y_i},$$
$$\forall\, x_1, \ldots, x_n \in \mathbb{R}, \quad \forall\, y_1, \ldots, y_m \in \{0, 1\}.$$

From Exercise 8.2, the log-likelihood function has at most one global maximum. So, if the MLE exists, it is unique.

Exercise 8.2. Let

$$h(x) := \frac{1}{1 + e^{-x}}, \qquad \forall\, x \in \mathbb{R}.$$

Fix $x \in \mathbb{R}$ and $y \in [0, 1]$. Define $t \colon \mathbb{R}^2 \to \mathbb{R}$ by

$$t(a, b) := \log\Big([h(ax + b)]^y [1 - h(ax + b)]^{1-y}\Big), \qquad \forall\, a, b \in \mathbb{R}.$$

Show that t is concave. Conclude that t has at most one global maximum.

Definition 8.3. A single variable **Logistic Regression** finds the values of $a, b \in \mathbb{R}$ that maximize the likelihood function $\ell(a, b)$. Then, given a new datapoint x, its probability of exhibiting the predicted trait (such as having a heart attack) is

$$h(ax + b).$$

Recall that $0 < h < 1$, so we can interpret $h(ax + b)$ as a probability. We can round this quantity to the nearest integer to perform a classification as well. That is, if $h(ax+b) > 1/2$, we predict that x exhibits the trait, and if $h(ax+b) \leq 1/2$, we predict that x does not exhibit the trait.

When $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^q$, we can similarly define a multivariable logistic regression that finds values $a \in \mathbb{R}^q$ and $b \in \mathbb{R}$ that maximizes the likelihood

$$\prod_{i=1}^{m} [h(\langle a, x^{(i)}\rangle + b)]^{y_i}[1 - h(\langle a, x^{(i)}\rangle + b)]^{1-y_i}.$$

Then, given a new datapoint $x \in \mathbb{R}^q$, its probability of exhibiting the predicted trait is

$$h(\langle a, x\rangle + b).$$

As before, we can round this quantity to the nearest integer to perform a classification.

Let's now apply logistic regression to classify emails as spam or ham. We already performed text classification in Exercise 4.14, so we can reuse some of that code.

The dataset we will use appears at the following link.

https://www.kaggle.com/datasets/uciml/sms-spam-collection-dataset

```
import numpy as np
import pandas as pd
import time as time
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn import metrics

df = pd.read_csv("spam.csv", encoding="latin-1")
df = df.drop(["Unnamed: 2", "Unnamed: 3", "Unnamed: 4"], axis=1)
df = df.rename(columns={"v1": "label", "v2": "text"})

# Preprocess text data
df["text"] = df["text"].str.lower()

dataset_train = df.iloc[0:4000, 1]
dataset_test = df.iloc[4000:, 1]

combined_data = df.iloc[:, 1]
labels_train = df.iloc[0:4000, 0]
labels_test = df.iloc[4000:, 0]

vectorizer = TfidfVectorizer(
    max_df = 0.5,
    min_df = 10,
    stop_words = "english",
)

t0 = time.time()
# we use a vectorizer on the entire dataset, otherwise
# the functions below will output errors
vector_data = vectorizer.fit_transform(combined_data)
print("Vectorized All Data in Time: %0.3f" % (time.time() - t0))
vector_data_train = vector_data[:len(dataset_train)]
vector_data_test = vector_data[len(dataset_train):]

t0 = time.time()

clf = LogisticRegression()
clf = clf.fit(vector_data_train, labels_train)
print("Logistic Fit in Time: %0.3f" % (time.time() - t0))

t0 = time.time()
pred = clf.predict(vector_data_test)
```

```
print("Prediction done in %0.3fs" % (time.time() - t0))
prediction_error = np.sum(pred != labels_test) / len(labels_test)

print("prediction error: %0.3f" % prediction_error)
```

We achieved a prediction error of .031 in less than 1/4 seconds.

```
Classification report for classifier:
              precision    recall  f1-score   support

         ham     0.9692    0.9963    0.9826      1360
        spam     0.9713    0.7972    0.8756       212

    accuracy                         0.9695      1572
   macro avg     0.9703    0.8967    0.9291      1572
weighted avg     0.9695    0.9695    0.9682      1572

[[1355    5]
 [  43  169]]
```

The function `predict` automatically converts the regression to classification by rounding the predicted probability to the nearest integer. That is, the following bit of code is equivalent to the last bit of code we used.

```
pred = clf.predict_proba(vector_data_test)
y_pred=[]
for i in range(len(pred)):
    if pred[i,0]>.5:
        y_pred.append('ham')
    else:
        y_pred.append('spam')
prediction_error = np.sum(y_pred != labels_test) / len(labels_test)
print("prediction error: %0.3f" % prediction_error)
```

If we change the cutoff value .5 to another number we get slightly better results. For example, with a value of .8, I got a classification error of .019.

For the sake of comparison, let's use a support vector classifier as we did in Exercise 4.14.

```
from scipy.stats import loguniform
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC

t0 = time.time()
param_grid = {
    "C": loguniform(1e3, 1e5),
    "gamma": loguniform(1e-4, 1e-1),
}
clf = RandomizedSearchCV(
    SVC(kernel = "rbf", class_weight = "balanced"), param_grid, n_iter=10
)
```

```
clf = clf.fit(vector_data_train, labels_train)
print("SVC Search Fit in Time: %0.3f" % (time.time() - t0))

t0 = time.time()
y_pred = clf.predict(vector_data_test)
print("Prediction done in %0.3fs" % (time.time() - t0))
prediction_error = np.sum(y_pred != labels_test) / len(labels_test)

print("prediction error: %0.3f" % prediction_error)
```

```
SVC Search Fit in Time: 14.756
Prediction done in 0.102s
prediction error: 0.023
```

This classifier uses a brute force search, hence the much slower run time, with a comparable prediction error.

9. Deep Learning, keras

9.1. **Handwriting Recognition and the MNIST Dataset.** Recall that in Exercise 4.17 we used support vector machines to classify handwritten digits, and we achieved about a 98% successful classification rate on the MNIST dataset. It turns out even better classification is possible with deep learning, which is a classical triumph in the field.

To make sure you have the right packages installed, run the following commands.

```
pip install --upgrade keras
pip install --upgrade tensorflow
```

Below, we will try to classify images from the MNIST dataset using neural networks. We will first define a neural network.

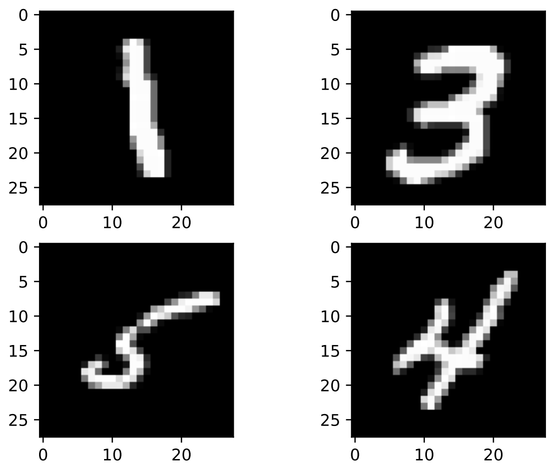


Figure 8. Four example images from the MNIST dataset

Definition 9.1 (**Feedforward Neural Network**). A **feedforward neural network** with k layers and activation function $h\colon \mathbb{R}\to\mathbb{R}$ is a function f defined as follows. Let $n_0,\ldots,n_{k-1}$ be positive integers. For each $1\leq i\leq k-1$, assume that

$$f_i\colon \mathbb{R}^{n_{i-1}}\to\mathbb{R}^{n_i},$$
$$f_k\colon \mathbb{R}^{n_{k-1}}\to\mathbb{R}.$$

Assume also that for all $1\leq i\leq k$, there exists $w^{(ij)}\in\mathbb{R}^{n_{i-1}}, t_{ij}\in\mathbb{R}$ such that the j^{th} component of f_i satisfies

$$f_{i,j}(x)=h(\langle w^{(ij)},x\rangle - t_{ij}),\qquad \forall\, x\in\mathbb{R}^{n_{i-1}}.$$

Then f is defined to be a function of the form

$$f:=f_k\circ f_{k-1}\circ\cdots\circ f_1.$$

We refer to $\max_{0\leq i\leq k-1} n_i$ as the **width** of the neural network. We also refer to k as the **depth** or **number of layers** of the neural network.

Note that if h is itself a linear function, then f is also a linear function. So, it is most sensible to choose a nonlinear activation function h.

Common examples of activation functions include:

- $h(s)=\mathrm{sign}(s)$ or $(1+\mathrm{sign}(s))/2$, $\forall\ s\in\mathbb{R}$ (Boolean activation function).
- $h(s)=\max(s,0)$, $\forall\ s\in\mathbb{R}$ (Rectified Linear Unit) (ReLU).
- $h(s)=(1+e^{-2s})^{-1}$, $\forall\ s\in\mathbb{R}$ (Sigmoid/Logistic Function)
- $h(s)=\tanh(s)=\frac{e^s-e^{-s}}{e^s+e^{-s}}$, $\forall\ s\in\mathbb{R}$.
- $h(s)=\frac{s}{1+e^{-s}}$, $\forall\ s\in\mathbb{R}$. (SiLU)

In the last case, note that $(1/2)[1+\tanh(s)]$ is equal to the sigmoid function.

Below are some other definitions we will need to understand the neural networks below. Let $\beta>0$. For any $x\in\mathbb{R}^n$, define the **softmax** function $p=p^{(\beta)}\colon\mathbb{R}^n\to\mathbb{R}^n$ by

$$p_i(x):=\frac{e^{\beta x_i}}{\sum_{j=1}^n e^{\beta x_j}},\qquad \forall\, x\in\mathbb{R}^n,\qquad \forall\, 1\leq i\leq n.$$

Observe that $\sum_{i=1}^n p_i=1$, so we can interpret p_i as an estimated probability that vector x is in class i. In the example below, the softmax function uses $n=10$ since there are ten digits (classes) to classify.

Let $(a_1,\ldots,a_n),(b_1,\ldots,b_n)$ be two nonnegative sequences of real numbers with $\sum_{i=1}^n a_i=\sum_{i=1}^n b_i=1$ and $b_i>0$ for all $1\leq i\leq n$. The **categorial cross entropy** of these sequences is $-\sum_{i=1}^n a_i\log b_i$. In the example below, $n=10$, $a_j=1$ when an image is labelled as digit j and $a_i=0$ for all $i\neq j$ with $0\leq i\leq 9$, and b_i is the currently estimated probability that the image is classified as digit i. (So $-\sum_{i=1}^n a_i\log b_i=-\log b_j$ in this case.) Finally, the quantity $\sum_{i=1}^n a_i\log b_i$ is averaged over all (60,000) images in the dataset to output the cross entropy. (More specifically, the loss function)

The **Adam** optimization method [KB15] (adaptive momentum estimation) is a variant of gradient descent that has been observed to perform much better in practice than gradient descent.

The **batch size** is the number of data points that are used when optimizing the loss function in a single epoch. Different epochs will load different sets of data points into the loss function.

In the code below, neural network weights are initialized to normal (standard gaussian) random variables (pseudorandom). The first argument in the `model.add(Dense(...))` command is the output dimension of that neural network layer. So, in this example, the output dimension matches the input dimension (they are both the number of pixels of a single image: 784).

The codes below were adapted from https://machinelearningmastery.com/ .

```
# Two Layer Neural Network for MNIST dataset
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.utils import to_categorical

# load data
(data_train, labels_train), (data_test, labels_test) = mnist.load_data()
# flatten each 28 by 28 image to a length 784 vector
data_train = data_train.reshape((data_train.shape[0], -1)).astype('float32')
data_test = data_test.reshape((data_test.shape[0], -1)).astype('float32')
# convert integer 0-255 grayscale values to real numbers between 0 and 1
data_train = data_train / 255
data_test = data_test / 255
# convert labels to binary vectors, as required by the cross entropy loss
labels_train = to_categorical(labels_train)
labels_test = to_categorical(labels_test)
num_classes = labels_test.shape[1]
num_pixels = data_train.shape[1]

# define neural network model
def neural_model():
    # create model, starting with the first input layer
    model = Sequential()
    model.add(Input(shape = (num_pixels,)))
    model.add(
        Dense(
            num_pixels,
            kernel_initializer = 'normal',
            activation = 'relu'
        )
    )
    model.add(
        Dense(
            num_classes,
            kernel_initializer = 'normal',
            activation='softmax'
        )
    )
    # Compile model
    model.compile(
        loss='categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return model

# build the model
model = neural_model()
# Fit the model.  verbose = 2 prints epoch progress
model.fit(
    data_train, labels_train, validation_data = (data_test, labels_test),
    epochs = 10,
    batch_size = 200
)
# Final evaluation of the model
scores = model.evaluate(data_test, labels_test, verbose = 0)
print("Two Layer Network Error: %.2f%%" % (100-scores[1]*100))
# Print a summary of the neural network
model.summary()
```

The final error I got was 2%, which matches the SVM performance we found in Exercise 4.17. (I ran the above code with only one layer instead of two, and I got a much worse error of 7.37%, though the code ran quite quickly.)

```
from sklearn import metrics
pred = np.argmax(model.predict(data_test), axis=-1)
(_, _), (_, labels_test) = mnist.load_data()

print(
    f"Classification report for classifier:\n"
    f"{metrics.classification_report(labels_test, pred, digits = 4)}\n"
    f"{metrics.confusion_matrix(labels_test, pred)}\n"
)
```

which outputs

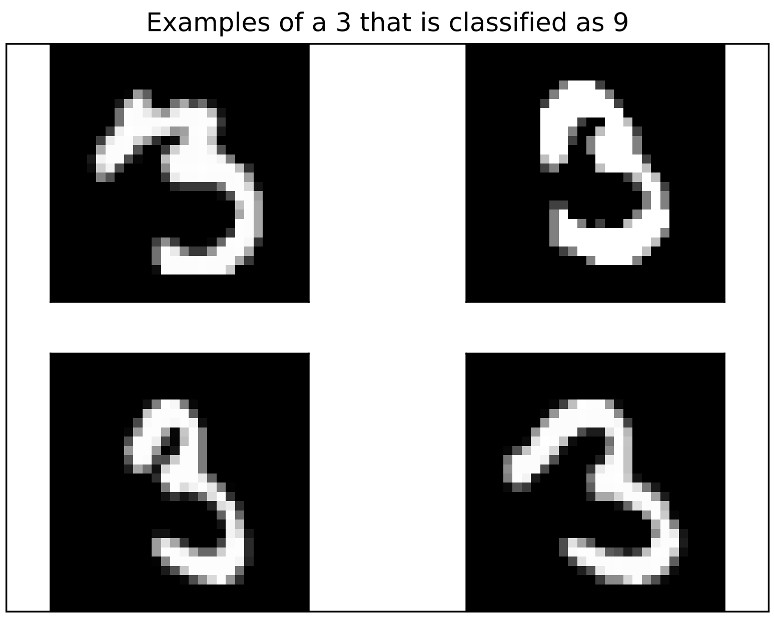
```
Classification report for classifier:
              precision    recall  f1-score   support

           0     0.9918    0.9827    0.9872       980
           1     0.9947    0.9850    0.9898      1135
           2     0.9759    0.9816    0.9787      1032
           3     0.9870    0.9752    0.9811      1010
           4     0.9867    0.9807    0.9837       982
           5     0.9788    0.9832    0.9810       892
           6     0.9804    0.9927    0.9865       958
           7     0.9797    0.9844    0.9820      1028
           8     0.9754    0.9784    0.9769       974
           9     0.9715    0.9792    0.9753      1009

    accuracy                         0.9823     10000
   macro avg     0.9822    0.9823    0.9822     10000
weighted avg     0.9823    0.9823    0.9823     10000
[[ 963    0    1    0    1    4    3    1    3    4]
 [    0 1118    4    1    0    1    3    1    7    0]
 [    2    1 1013    1    1    0    2    6    5    1]
 [    0    0    3  985    0    6    0    6    0   10]
 [    1    0    4    1  963    0    5    2    0    6]
 [    2    0    0    5    0  877    3    1    3    1]
 [    0    2    1    1    1    2  951    0    0    0]
 [    0    1    9    0    0    0    0 1012    3    3]
 [    2    0    3    2    3    3    2    2  953    4]
 [    1    2    0    2    7    3    1    2    3  988]]
```

The last matrix C (known as a confusion matrix) is defined so that entry C_{ij} is the number of observations known to be in class i and predicted to be in class j (in this case $0 \leq i, j \leq 9$.) So, e.g. $C_{3,9} = 10$ means there are ten examples of digit 3 that were (mistakenly) classified as digit 9.

We can view some mis-classified images with the following code.

```
import matplotlib.pyplot as plt
import numpy as np

def plot_mislabels(label_0, label_1, data_test,
                   labels_test, pred, num_images = 4):
    # plots num_images instances of an image in
    #  class label_0, that has been classified as label_1

    mis_class = np.intersect1d(
        np.where(labels_test == label_0),
        np.where(pred == label_1)
    )

    plt.title('Examples of a ' + str(label_0)
              + ' that is classified as ' + str(label_1))
    plt.xticks([])
    plt.yticks([])
    for i in range(num_images):
        plt.subplot(2, 2, i + 1)
        plt.imshow(data_test[mis_class[i]], cmap = plt.get_cmap('gray'))
        plt.xticks([])
        plt.yticks([])
    plt.savefig('mis_class_1.pdf')
    plt.show()
```

```
(data_train, labels_train), (data_test, labels_test) = mnist.load_data()
plot_mislabels(3, 9, data_test, labels_test, pred)
```



One disadvantage of neural networks is they have give no obvious explanation for how they work. To try to understand which pixels of an image play a role in their classification, we can go through each pixel of each image in the test set, to see if the predicted digit changes when each pixel is changed to either a 0 or 1 value. (Recall that each image pixel consists of a grayscale value in $[0, 1]$.)

```
from tqdm import tqdm
heatmap = np.zeros(num_pixels)
#small_shift = 1/10
for i in tqdm(range(num_pixels)):
    test_copy1 = data_test.copy()
    test_copy2 = data_test.copy()
    test_copy1[:, i] = 1
    test_copy2[:, i] = 0
    pred1 = np.argmax(model.predict(test_copy1, verbose = 0), axis=-1)
    pred2 = np.argmax(model.predict(test_copy2, verbose = 0), axis=-1)
    heatmap[i] = np.mean(np.logical_or(pred != pred1, pred != pred2))

heatmap = heatmap.reshape([28, 28])
plt.imshow(heatmap, cmap = "jet")
plt.title(
    '''Pixels where changing their value
    to 1 or 0 changes the prediction'''
)
plt.colorbar(
    label = '''Fraction of images in training set
             where the prediction was changed'''
)
plt.xticks([])
plt.yticks([])
plt.savefig('heatmap0.pdf')
plt.show()
```

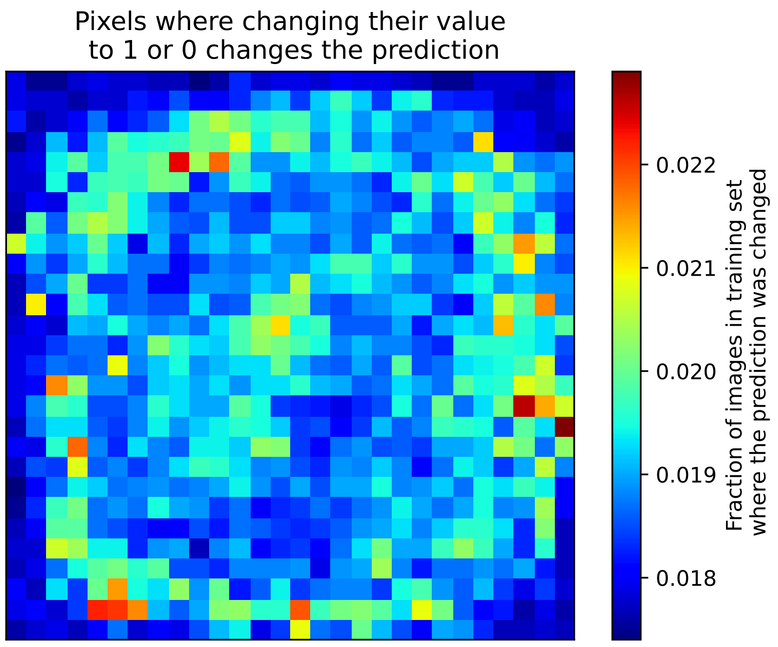


FIGURE 9. Heatmap for the network with two layers.

In the above neural network approach, we treat each image a vector, i.e. we don't really use any information about pixels that are adjacent to each other (in a straightforward way). In contrast, our approach below will be designed to use information about adjacent pixels. More specifically, we will use a convolutional neural network. That is, we will take averages of blocks of pixel values when designing the neural network.

Recall that you have seen convolution before, when finding the distribution of the sum of two independent random variables. Let X, Y be independent integer-valued random variables. Let $t \in \mathbb{Z}$. Then

$$\mathbf{P}(X+Y=t)=\sum_{j\in\mathbb{Z}}\mathbf{P}(X=j)\mathbf{P}(Y=t-j),$$

which is the convolution (on the integers) of the PMFs of X and Y. Similarly, if X, Y are independent continuous random variables with PDFs f_X and f_Y respectively, then the PDF

f_{X+Y} of $X+Y$ is the convolution of f_X and f_Y.

$$f_{X+Y}(t) = (f_X * f_Y)(t) = \int_{\mathbb{R}} f_X(x) f_Y(t-x)dx, \qquad \forall\, t \in \mathbb{R}.$$

We can understand convolution on the integers or on the real line as taking an average of one function using another function. Likewise, the discrete two-dimensional convolution we use below can be understood as a two-dimensional average of pixel values. The first argument of the two-dimensional convolutional layer `Conv2D(...)` is the output dimension, which is set to be 32. That is, after the convolutional layer is applied, we take 32 different copies of its output with 32 different weights. The second argument is the size of the "convolution window." In the code below, the layer takes a weighted average of several 5 by 5 blocks of the 784 pixel image. Since the original image is 28 by 28 and the average is taken over all possible 5 by 5 squares in the image, the output of the convolutional layer are 24 by 24 images (noting that $28-5+1=24$, i.e. there are 24 ways to put a 5 by 5 square in a 28 by 28 grid.) The next max-pooling layer takes the maximum value over four values in a grid of disjoint 2 by 2 blocks. Next, the dropout layer randomly sets to zero each of its inputs with given probability (which in this case is .2). The dropout layer is meant to protect against overfitting.

As mentioned above, a convolutional layer directly uses information about pixels that are near each other in the image, whereas our previous neural network approach did not explicitly use this information. Moreover, our implementation of a convolutional layer is somewhat translation invariant, which should aid in detecting features that do not depend on their exact position in the image.

```
# Simple CNN (convolutional neural network) for the MNIST Dataset
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.utils import to_categorical
# load data
(data_train, labels_train), (data_test, labels_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
data_train = data_train.reshape((data_train.shape[0], 28, 28, 1)).astype('float32')
data_test = data_test.reshape((data_test.shape[0], 28, 28, 1)).astype('float32')
# convert integer 0-255 grayscale values to real numbers between 0 and 1
data_train = data_train / 255
data_test = data_test / 255
# convert labels to binary vectors, as required by the cross entropy loss
labels_train = to_categorical(labels_train)
labels_test = to_categorical(labels_test)
num_classes = labels_test.shape[1]
# define a simple CNN (convolutional neural network) model
def cnn_model():
    # create model
    model = Sequential()
    model.add(Input(shape = (28, 28, 1)))
    model.add(
        Conv2D(
            32,
            (5, 5),
            activation = 'relu')
        )
    model.add(MaxPooling2D())
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation = 'relu'))
    model.add(Dense(num_classes, activation = 'softmax'))
    # Compile model
    model.compile(
        loss = 'categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return model
# build the model
model = cnn_model()
# Fit the model
model.fit(
    data_train, labels_train, validation_data = (data_test, labels_test),
    epochs = 10,
    batch_size = 200
)
# Final evaluation of the model
scores = model.evaluate(data_test, labels_test, verbose = 0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
# Print a summary of the neural network
model.summary()
```

Here is the output of the program:
CNN Error: 1.11%

Layer (type)	Output Shape	Number of Parameters
conv2d (Conv2D)	(None, 24, 24, 32)	832
max pooling 2d (MaxPooling2D)	(None, 12, 12, 32)	0
dropout (Dropout)	(None, 12, 12, 32)	0
flatten (Flatten)	(None, 4608)	0
dense (Dense)	(None, 128)	589,952
dense 1 (Dense)	(None, 10)	1,290

Total params: 1,776,224 (6.78 MB)
Trainable params: 592,074 (2.26 MB)
Non-trainable params: 0 (0.00 B)

Optimizer params: 1,184,150 (4.52 MB)

As we can see, we improved upon the classification error of our earlier approaches.

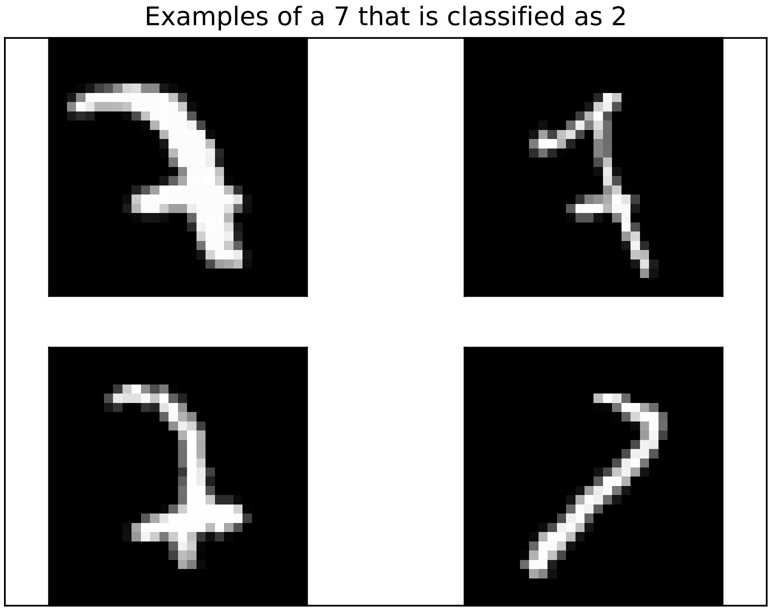
```
Classification report for classifier:
              precision    recall  f1-score   support

           0     0.9928    0.9847    0.9887       980
           1     0.9956    0.9877    0.9916      1135
           2     0.9930    0.9641    0.9784      1032
           3     0.9737    0.9891    0.9813      1010
           4     0.9919    0.9919    0.9919       982
           5     1.0000    0.9159    0.9561       892
           6     0.9916    0.9802    0.9858       958
           7     0.9950    0.9728    0.9838      1028
           8     0.8565    0.9990    0.9223       974
           9     0.9889    0.9673    0.9780      1009

    accuracy                         0.9759     10000
   macro avg     0.9779    0.9753    0.9758     10000
weighted avg     0.9782    0.9759    0.9763     10000

[[ 965    0    0    0    0    0    3    1   11    0]
 [    0 1121    1    2    0    0    1    0   10    0]
 [    1    0  995    0    1    0    0    3   32    0]
 [    0    0    2  999    0    0    0    0    9    0]
 [    0    0    0    0  974    0    0    0    3    5]
 [    2    0    0   19    0  817    4    0   49    1]
 [    2    1    0    0    1    0  939    0   15    0]
 [    0    2    4    3    1    0    0 1000   13    5]
 [    1    0    0    0    0    0    0    0  973    0]
 [    1    2    0    3    5    0    0    1   21  976]]
```

We can use our function from before to output some mis-classified images.

```
(_, _), (data_test, labels_test) = mnist.load_data()
plot_mislabels(7, 2, data_test, labels_test, pred)
```

Once again, we can plot a heatmap to try to visualize how the network classifies the images.

```
from tqdm import tqdm
heatmap = np.zeros(num_pixels)
#small_shift = 1/10
for i in tqdm(range(num_pixels)):
    test_copy1 = data_test.copy()
    test_copy2 = data_test.copy()
    test_copy1[:, i // 28, i % 28] = 1
    test_copy2[:, i // 28, i % 28] = 0
```



```
    pred1 = np.argmax(model.predict(test_copy1, verbose = 0), axis=-1)
    pred2 = np.argmax(model.predict(test_copy2, verbose = 0), axis=-1)
    heatmap[i] = np.mean(np.logical_or(pred != pred1, pred != pred2))

heatmap = heatmap.reshape([28, 28])
plt.imshow(heatmap, cmap = "jet")
plt.title(
    '''Pixels where changing their value
    to 1 or 0 changes the prediction'''
)
plt.colorbar(
    label = '''Fraction of images in training set
             where the prediction was changed'''
)
plt.xticks([])
plt.yticks([])
plt.savefig('heatmap0.pdf')
plt.show()
```

In this final example, we will add another convolutional layer to the network.

```
# Larger CNN for the MNIST Dataset
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
```

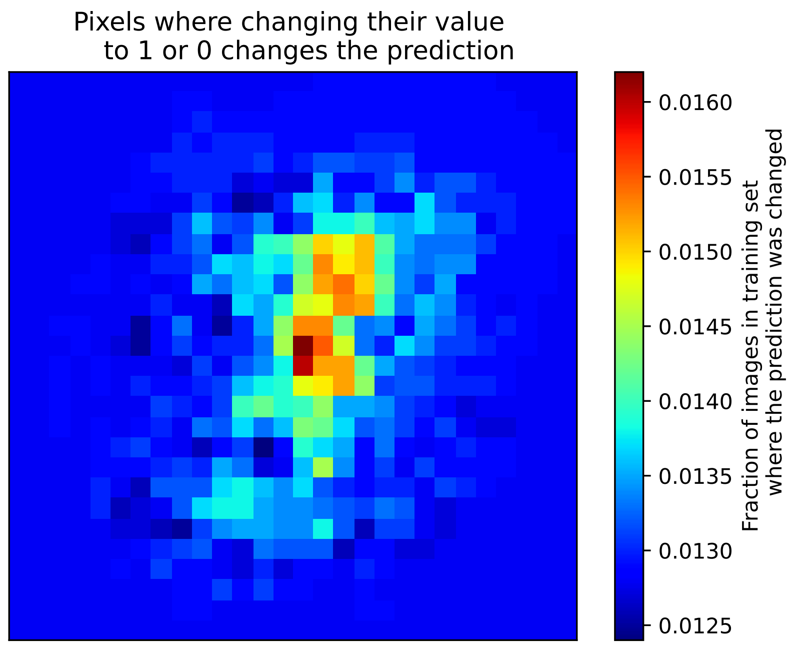


FIGURE 10. Heatmap for the network with one convolutional layer.

```
from tensorflow.keras.utils import to_categorical
# load data
(data_train, labels_train), (data_test, labels_test) = mnist.load_data()
# reshape to be [samples][width][height][channels]
data_train = data_train.reshape((data_train.shape[0], 28, 28, 1)).astype('float32')
data_test = data_test.reshape((data_test.shape[0], 28, 28, 1)).astype('float32')
# convert integer 0-255 grayscale values to real numbers between 0 and 1
data_train = data_train / 255
data_test = data_test / 255
# convert labels to binary vectors, as required by the cross entropy loss
labels_train = to_categorical(labels_train)
labels_test = to_categorical(labels_test)
num_classes = labels_test.shape[1]
# define the larger model
def larger_model():
    # create model
    model = Sequential()
    model.add(Input(shape = (28, 28, 1)))
    model.add(
        Conv2D(
            30,
            (5, 5),
            activation = 'relu'
        )
    )
    model.add(MaxPooling2D())
    model.add(
        Conv2D(
            15,
            (3, 3),
            activation = 'relu'
        )
    )
    model.add(MaxPooling2D())
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation = 'relu'))
    model.add(Dense(50, activation = 'relu'))
    model.add(Dense(num_classes, activation = 'softmax'))
    # Compile model
    model.compile(
        loss = 'categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return model
# build the model
model = larger_model()
# Fit the model
model.fit(
    data_train, labels_train, validation_data = (data_test, labels_test),
    epochs = 10,
    batch_size = 200
)
# Final evaluation of the model
scores = model.evaluate(data_test, labels_test, verbose = 0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100))
# Print a summary of the neural network
model.summary()
```

Large CNN Error: 0.77%
Total params: 179,801 (702.35 KB)
Trainable params: 59,933 (234.11 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 119,868 (468.24 KB)
Running our above code for a classification report outputs:

```
Classification report for classifier:
             precision    recall  f1-score   support
```

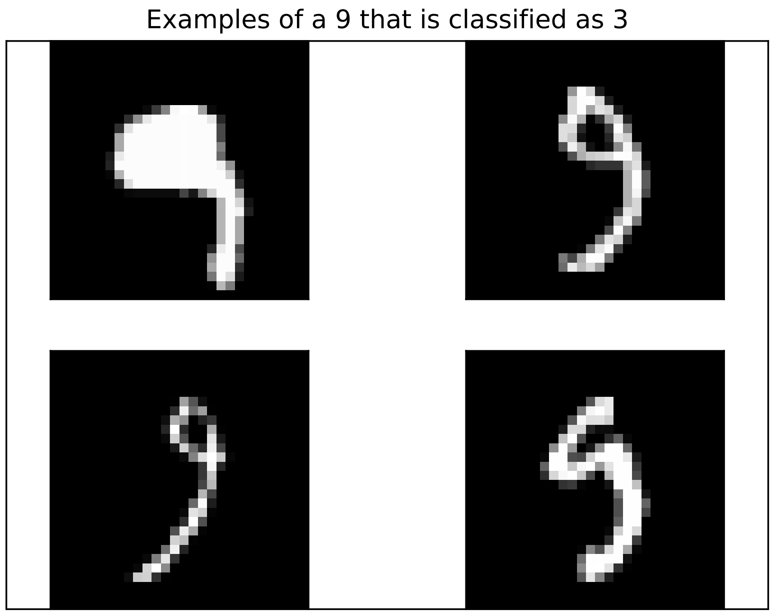
Layer (type)	Output Shape	Number of Parameters
conv2d (Conv2D)	(None, 24, 24, 30)	780
max pooling 2d (MaxPooling2D)	(None, 12, 12, 30)	0
conv2d 1 (Conv2D)	(None, 10, 10, 15)	4,065
max pooling2d 1 (MaxPooling2D)	(None, 5, 5, 15)	0
dropout (Dropout)	(None, 5, 5, 15)	0
flatten (Flatten)	(None, 375)	0
dense (Dense)	(None, 128)	48,128
dense 1 (Dense)	(None, 50)	6,450
dense 2 (Dense)	(None, 10)	510

```
           0     0.9829    0.9980    0.9904       980
           1     0.9973    0.9903    0.9938      1135
           2     0.9885    0.9981    0.9932      1032
           3     0.9891    0.9921    0.9906      1010
           4     0.9969    0.9878    0.9923       982
           5     0.9921    0.9888    0.9905       892
           6     0.9865    0.9937    0.9901       958
           7     0.9903    0.9903    0.9903      1028
           8     0.9846    0.9877    0.9862       974
           9     0.9939    0.9762    0.9850      1009

    accuracy                         0.9903     10000
   macro avg     0.9902    0.9903    0.9902     10000
weighted avg     0.9903    0.9903    0.9903     10000

[[ 978    0    0    0    0    0    1    1    0    0]
 [    4 1124    2    1    0    1    2    0    1    0]
 [    0    0 1030    0    0    0    0    1    1    0]
 [    1    0    1 1002    0    3    0    1    2    0]
 [    0    0    0    0  970    0    5    0    3    4]
 [    1    0    0    5    0  882    2    1    1    0]
 [    3    1    0    0    1    1  952    0    0    0]
 [    1    0    7    0    0    0    0 1018    1    1]
 [    6    0    1    0    0    0    3    1  962    1]
 [    1    2    1    5    2    2    0    5    6  985]]
```

We can use our function from before to output some mis-classified images.

```
 (_, _), (data_test, labels_test) = mnist.load_data()
plot_mislabels(7, 2, data_test, labels_test, pred)
```

And we can again plot a heatmap for influential pixels.



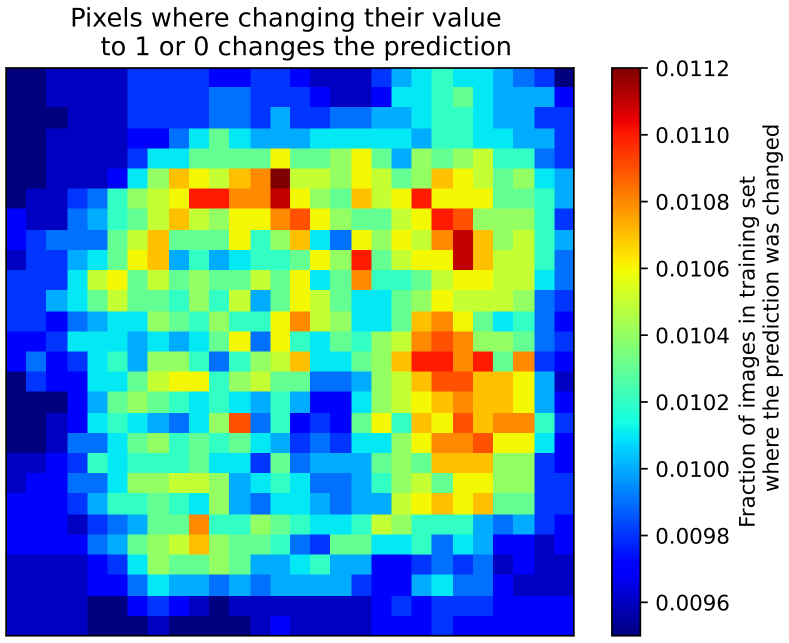


FIGURE 11. Heatmap for the network with two convolutional layers.

Exercise 9.2. Adjust the parameters such as batch size for the deep learning approaches to classifying digits in the MNIST dataset. Are you able to significantly improve the percentage of correct classifications above 99.3% ?

Exercise 9.3. Use a convolutional neural network to classify the CIFAR 10 and CIFAR 100 datasets found here:
https://www.cs.toronto.edu/∼kriz/cifar.html
https://keras.io/api/datasets/cifar10/
https://keras.io/api/datasets/cifar100/
What error rate can you obtain? For CIFAR-10, I would consider an error rate below 20% to be quite good.

For a simpler approach, I also used a single layer network (which would be almost identical to logistic regression), and it obtained a 7% error rate.

There are many pretrained neural networks for image classification, such as ResNet50. Below, we apply ResNet50 to image classification with the MNIST dataset, for illustrative purposes. However, the performance is quite slow and not so good, perhaps partly because ResNet50 is designed for much larger images than those from the MNIST dataset.

```
# ResNet50 PreTrained Network for Image Classification
import keras
from keras.applications.resnet50 import ResNet50
from keras.applications.resnet50 import preprocess_input, decode_predictions
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Resizing
from tensorflow.keras.utils import to_categorical
import tensorflow as tf
import numpy as np

# load data
(data_train, labels_train), (data_test, labels_test) = mnist.load_data()
# expand new axis, channel axis
data_train = np.expand_dims(data_train, axis=-1)
data_test = np.expand_dims(data_test, axis=-1)
# need 3 channel (instead of 1)
data_train = np.repeat(data_train, 3, axis=-1)
data_test = np.repeat(data_test, 3, axis=-1)
# it's always better to normalize
data_train = data_train.astype('float32') / 255
data_test = data_test.astype('float32') / 255
# resize the input shape , i.e. old shape: 28, new shape: 32
data_train = tf.image.resize(data_train, [32,32]) # if we want to resize
data_test = tf.image.resize(data_test, [32,32]) # if we want to resize

# convert labels to binary vectors, as required by the cross entropy loss
labels_train = to_categorical(labels_train)
labels_test = to_categorical(labels_test)
num_classes = labels_test.shape[1]

# define neural network model
```

```
def neural_model():
    # create model, starting with the first input layer
    model = Sequential()
    model.add(
        ResNet50(
            include_top = False,
            pooling = "avg",
            input_shape=(32, 32, 3)
        )
    )
    model.add(
        Dense(
            num_classes,
            kernel_initializer = 'normal',
            activation = 'softmax'
        )
    )

    # Say not to train third layer (ResNet) model as it is already trained
    model.layers[0].trainable = False
    # Compile model
    model.compile(
        loss='categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return model

# build the model
model = neural_model()
# Fit the model.
model.fit(
    data_train, labels_train, validation_data = (data_test, labels_test),
    epochs = 10,
    batch_size = 200
)
# Final evaluation of the model
scores = model.evaluate(data_test, labels_test, verbose = 0)
print("ResNet50 Error: %.2f%%" % (100-scores[1]*100))
# Print a summary of the neural network
model.summary()
```

ResNet50 Error: 7.72%

9.2. **Facial Recognition.**

```
import numpy as np
from sklearn.datasets import fetch_lfw_people
from sklearn import metrics

lfw_people = fetch_lfw_people(min_faces_per_person = 70, resize = 0.4)
test_size = 200

# lfw_people.images is a 3-dimensional array, consisting of n_samples of images,
# where each image has width w and height h, in pixels.  the greyscale value of each
# image is a real number between 0 and 1
n_samples, height, width = lfw_people.images.shape
train_images = lfw_people.images[test_size:1 + n_samples, :, :]
test_images = lfw_people.images[0:test_size, :, :]

# the label to predict is the ID of the person
y = lfw_people.target
target_names = lfw_people.target_names
num_classes = target_names.shape[0]
labels_train = y[test_size:1 + n_samples]
labels_test = y[0:test_size]

# put the image data into a 2-dimensional array, where each row of the matrix
# corresponds to a distinct image
data_train = train_images.reshape(train_images.shape[0], -1)
data_test = test_images.reshape(test_images.shape[0], -1)
num_pixels = data_train.shape[1]

labels_train = to_categorical(labels_train)
labels_test = to_categorical(labels_test)

def neural_model():
    model = Sequential()
    model.add(Input(shape = (num_pixels,)))
    model.add(
        Dense(
            num_pixels,
            kernel_initializer = 'normal',
            activation = 'relu'
        )
    )
    model.add(
        Dense(
            num_classes,
            kernel_initializer = 'normal',
            activation = 'softmax'
        )
    )
    model.compile(
        loss = 'categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return model

model = neural_model()
model.fit(
    data_train,
    labels_train_cat,
    validation_data = (data_test, labels_test),
    epochs = 20,
    batch_size = 50
)

scores = model.evaluate(data_test, labels_test, verbose = 0)
print("Two Layer Network Error: %.2f%%" % (100-scores[1]*100))
model.summary()
pred = np.argmax(model.predict(data_test), axis=-1)
labels_test = np.where(labels_test == 1)[1]
labels_train = np.where(labels_train == 1)[1]

print(
    f"Classification report for classifier:\n"
    f"{metrics.classification_report(labels_test, pred, digits = 4)}\n"
    f"{metrics.confusion_matrix(labels_test, pred)}\n"
)
```

This has output

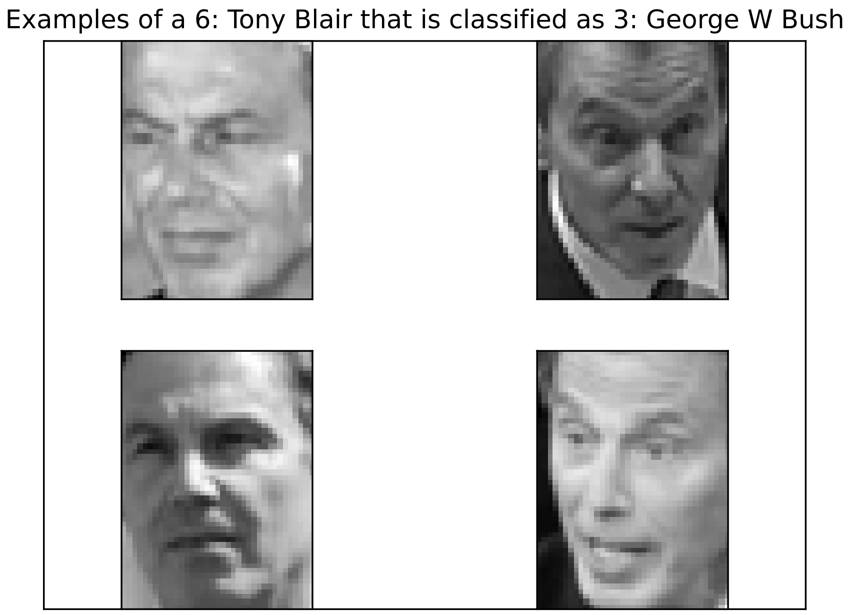
```
Two Layer Network Error: 21.50%
Model: "sequential_12"
```

Layer (type)	**Output Shape**	**Param #**
dense_25 (Dense)	(None, 1850)	3,424,350
dense_26 (Dense)	(None, 7)	12,957

```
 Total params: 10,311,923 (39.34 MB)
 Trainable params: 3,437,307 (13.11 MB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 6,874,616 (26.22 MB)
Classification report for classifier:
              precision    recall  f1-score   support

           0     0.6500    0.8125    0.7222        16
           1     0.9259    0.6579    0.7692        38
           2     1.0000    0.6500    0.7879        20
           3     0.8313    0.9200    0.8734        75
           4     0.5926    0.8889    0.7111        18
           5     1.0000    0.4545    0.6250        11
           6     0.6400    0.7273    0.6809        22

    accuracy                         0.7850       200
   macro avg     0.8057    0.7302    0.7385       200
weighted avg     0.8184    0.7850    0.7835       200

[[13  0  0  1  1  0  1]
 [ 4 25  0  4  1  0  4]
 [ 2  1 13  3  1  0  0]
 [ 0  1  0 69  3  0  2]
 [ 0  0  0  1 16  0  1]
 [ 0  0  0  1  4  5  1]
 [ 1  0  0  4  1  0 16]]
```

As before, we can plot some images that were not classified well.

```
import matplotlib.pyplot as plt
import numpy as np

label_0 = 6
label_1 = 3

lfw_people = fetch_lfw_people(min_faces_per_person = 70, resize = 0.4)
data_test = lfw_people.images[0:test_size, :, :]
labels_train = y[test_size:1 + n_samples]
labels_test = y[0:test_size]

mis_class = np.intersect1d(
    np.where(labels_test == label_0),
    np.where(pred == label_1)
)

plt.title('Examples of a ' + str(label_0) + ': ' + str(target_names[label_0])
          +' that is classified as ' + str(label_1) + ': ' + str(target_names[label_1])
)
plt.xticks([])
plt.yticks([])
for i in range(4):
    plt.subplot(2, 2, i + 1)
    plt.imshow(data_test[mis_class[i]], cmap=plt.get_cmap('gray'))
    plt.xticks([])
    plt.yticks([])
plt.savefig('face_examples0.pdf')
plt.show()
```

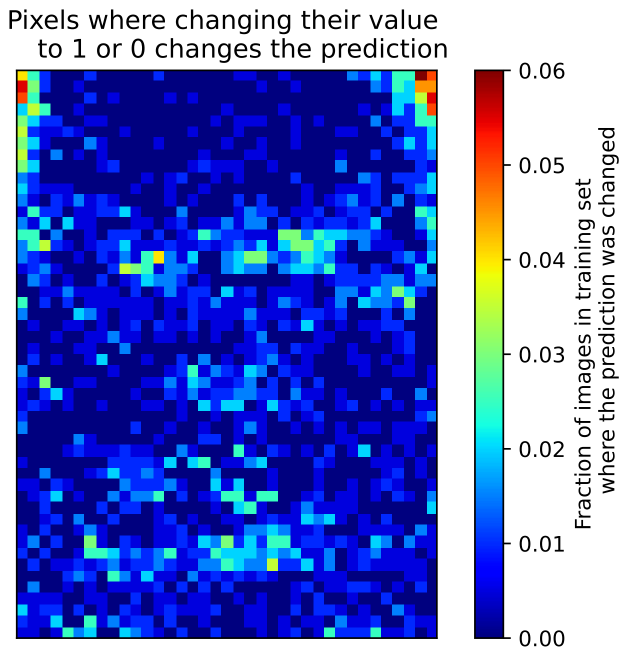

Examples of a 6: Tony Blair that is classified as 3: George W Bush

We can see these example images are not centered, so it is reasonable that our classifier did not perform well for these images.

We can again plot a heatmap.

```
from tqdm import tqdm
num_pixels = height * width
heatmap = np.zeros(num_pixels)
data_train = train_images.reshape(train_images.shape[0], -1)
data_test = test_images.reshape(test_images.shape[0], -1)
#small_shift = 1/10
for i in tqdm(range(num_pixels)):
    test_copy1 = data_test.copy()
    test_copy2 = data_test.copy()
    test_copy1[:, i] = 1
    test_copy2[:, i] = 0
    pred1 = np.argmax(model.predict(test_copy1, verbose = 0), axis=-1)
    pred2 = np.argmax(model.predict(test_copy2, verbose = 0), axis=-1)
    heatmap[i] = np.mean(np.logical_or(pred != pred1, pred != pred2))

heatmap = heatmap.reshape([height, width])
plt.imshow(heatmap, cmap = "jet")
plt.title(
    '''Pixels where changing their value
    to 1 or 0 changes the prediction'''
)
```

```
plt.colorbar(
    label = '''Fraction of images in training set
              where the prediction was changed'''
)
plt.xticks([])
plt.yticks([])
plt.savefig('heatmap0.pdf')
plt.show()
```



Finally, we apply a neural network with two convolutional layers

```
import numpy as np
from sklearn.datasets import fetch_lfw_people
from sklearn import metrics

lfw_people = fetch_lfw_people(min_faces_per_person = 70, resize = 0.4)
test_size = 200

# lfw_people.images is a 3-dimensional array, consisting of n_samples of images,
# where each image has width w and height h, in pixels.  the greyscale value of each
# image is a real number between 0 and 1
n_samples, height, width = lfw_people.images.shape
data_train = lfw_people.images[test_size:1+n_samples, :, :]
data_test = lfw_people.images[0:test_size, :, :]

# the label to predict is the ID of the person
```

```
y = lfw_people.target
target_names = lfw_people.target_names
num_classes = target_names.shape[0]
labels_train = y[test_size:1 + n_samples]
labels_test = y[0:test_size]

num_pixels = data_train.shape[1]

# Larger CNN for the MNIST Dataset
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.utils import to_categorical

# reshape to be [samples][width][height][channels]
data_train = data_train.reshape((data_train.shape[0], height, width, 1)).astype('float32
data_test = data_test.reshape((data_test.shape[0], height, width, 1)).astype('float32')
# convert labels to binary vectors, as required by the cross entropy loss
labels_train = to_categorical(labels_train)
labels_test = to_categorical(labels_test)
num_classes = labels_test.shape[1]
# define the larger model
def larger_model():
    # create model
    model = Sequential()
    model.add(Input(shape = (height, width, 1)))
    model.add(
        Conv2D(
            30,
            (5, 5),
            activation = 'relu'
        )
    )
    model.add(MaxPooling2D())
    model.add(
        Conv2D(
            15,
            (3, 3),
            activation = 'relu'
        )
    )
    model.add(MaxPooling2D())
    model.add(Dropout(0.2))
    model.add(Flatten())
    # added another dense layer compared to previous approach
    model.add(Dense(128, activation = 'relu'))
    model.add(Dense(50, activation = 'relu'))
    model.add(Dense(num_classes, activation = 'softmax'))
    # Compile model
    model.compile(
        loss = 'categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return model
# build the model
model = larger_model()
# Fit the model
model.fit(
    data_train, labels_train, validation_data = (data_test, labels_test),
    epochs = 20,
    batch_size = 50
)
# Final evaluation of the model
scores = model.evaluate(data_test, labels_test, verbose = 0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100))
# Print a summary of the neural network
model.summary()
```

```
Large CNN Error: 16.00%
```

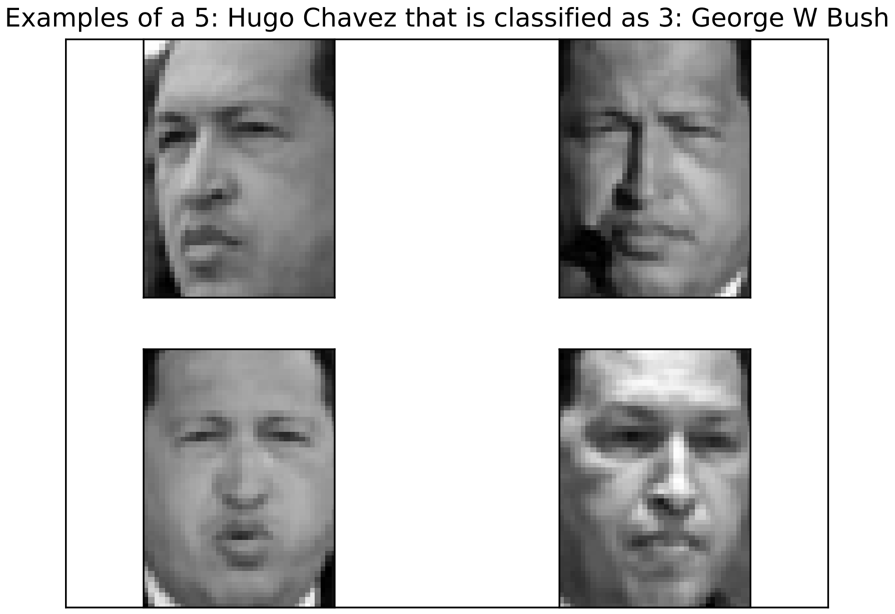
Layer (type)	**Output Shape**	**Param #**
conv2d_6 (Conv2D)	(None, 46, 33, 30)	780
max_pooling2d_6 (MaxPooling2D)	(None, 23, 16, 30)	0
conv2d_7 (Conv2D)	(None, 21, 14, 15)	4,065
max_pooling2d_7 (MaxPooling2D)	(None, 10, 7, 15)	0
dropout_4 (Dropout)	(None, 10, 7, 15)	0
flatten_4 (Flatten)	(None, 1050)	0
dense_27 (Dense)	(None, 128)	134,528
dense_28 (Dense)	(None, 50)	6,450
dense_29 (Dense)	(None, 7)	357

```
 Total params: 438,542 (1.67 MB)
 Trainable params: 146,180 (571.02 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 292,362 (1.12 MB)
```

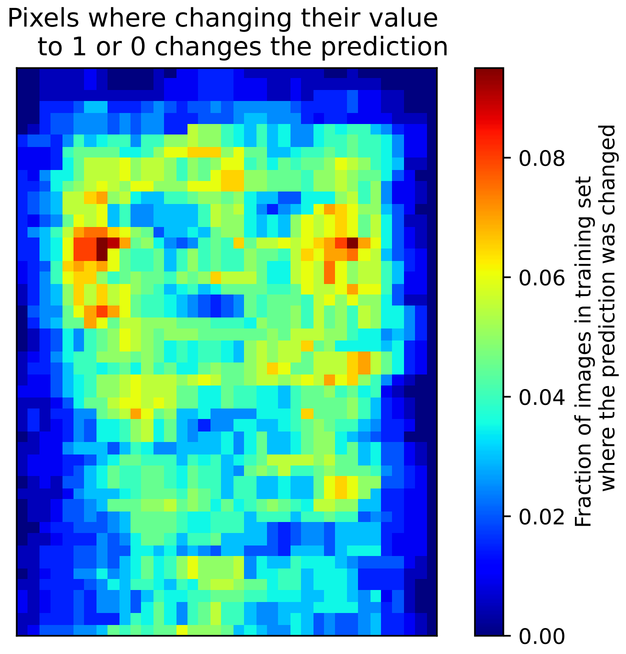
```
pred = np.argmax(model.predict(data_test), axis=-1)
lfw_people = fetch_lfw_people(min_faces_per_person = 70, resize = 0.4)
labels_test = y[0:test_size]


print(
    f"Classification report for classifier:\n"
    f"{metrics.classification_report(labels_test, pred, digits = 4)}\n"
    f"{metrics.confusion_matrix(labels_test, pred)}\n"
)
```

```
Classification report for classifier:
              precision    recall  f1-score   support

           0     1.0000    0.5625    0.7200        16
           1     0.8684    0.8684    0.8684        38
           2     0.7391    0.8500    0.7907        20
           3     0.9231    0.9600    0.9412        75
           4     0.6875    0.6111    0.6471        18
           5     0.7273    0.7273    0.7273        11
           6     0.8400    0.9545    0.8936        22

    accuracy                         0.8550       200
   macro avg     0.8265    0.7906    0.7983       200
weighted avg     0.8593    0.8550    0.8511       200

[[ 9  0  2  0  3  0  2]
 [ 0 33  2  1  1  1  0]
 [ 0  2 17  1  0  0  0]
 [ 0  0  1 72  1  1  0]
 [ 0  3  1  2 11  0  1]
 [ 0  0  0  2  0  8  1]
 [ 0  0  0  0  0  1 21]]
```

```
import matplotlib.pyplot as plt
import numpy as np

label_0 = 5
label_1 = 3

lfw_people = fetch_lfw_people(min_faces_per_person = 70, resize = 0.4)
data_test = lfw_people.images[0:test_size, :, :]

mis_class = np.intersect1d(np.where(labels_test == label_0), np.where(pred == label_1))
print(mis_class)

plt.title('Examples of a ' + str(label_0) + ': ' + str(target_names[label_0])
         +' that is classified as ' + str(label_1) + ': ' + str(target_names[label_1])
)
plt.xticks([])
plt.yticks([])
for i in range(4):
    plt.subplot(2, 2, i + 1)
    plt.imshow(data_test[mis_class[i]], cmap=plt.get_cmap('gray'))
    plt.xticks([])
    plt.yticks([])
plt.savefig('face_examples2.pdf')
plt.show()
```


Examples of a 5: Hugo Chavez that is classified as 3: George W Bush

```
from tqdm import tqdm
num_pixels = height * width
heatmap = np.zeros(num_pixels)
for i in tqdm(range(num_pixels)):
    test_copy1 = data_test.copy()
    test_copy2 = data_test.copy()
    test_copy1[:, i // height, i % width] = 1
    test_copy2[:, i // height, i % width] = 0
    pred1 = np.argmax(model.predict(test_copy1, verbose = 0), axis=-1)
    pred2 = np.argmax(model.predict(test_copy2, verbose = 0), axis=-1)
    heatmap[i] = np.mean(np.logical_or(pred != pred1, pred != pred2))

heatmap = heatmap.reshape([height, width])
plt.imshow(heatmap, cmap = "jet")
plt.title(
    '''Pixels where changing their value
    to 1 or 0 changes the prediction'''
)
plt.colorbar(
    label = '''Fraction of images in training set
             where the prediction was changed'''
)
plt.xticks([])
plt.yticks([])
plt.savefig('heatmap0.pdf')
plt.show()
```



9.3. **Document Classification.** We will now recycle some of our above code to classify documents using a two-layer neural network. Recall that our best previous error was around 45%. In the following approach, we can achieve around 17% with six epochs.

```
import numpy as np
import itertools
import math
import time as time

t0 = time.time()
from sklearn.datasets import fetch_20newsgroups
from sklearn.cluster import KMeans
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

categories = [
    "comp.graphics",
    "misc.forsale",
    "rec.sport.baseball",
    "sci.space",
    "talk.politics.misc",
    "talk.religion.misc",
]

# import data, remove extraneous bits of text
dataset_train = fetch_20newsgroups(
    remove = ("headers", "footers", "quotes"),
    subset = "train",
    categories = categories,
    shuffle = True,
    random_state = 42,
)
dataset_test = fetch_20newsgroups(
    remove = ("headers", "footers", "quotes"),
    subset = "test",
    categories = categories,
    shuffle = True,
    random_state = 42,
)

vectorizer = TfidfVectorizer(
    max_df = 0.5,
    min_df = 10,
    stop_words = "english",
)

combined_data = dataset_train.data + dataset_test.data
labels_train = dataset_train.target

t0 = time.time()
# we use a vectorizer on the entire dataset, otherwise
# the functions below will output errors
vector_data = vectorizer.fit_transform(combined_data)
print("Vectorized All Data in Time: %0.3f" % (time.time() - t0))
vector_data_train = vector_data[:len(dataset_train.data)]
vector_data_test = vector_data[len(dataset_train.data):]
num_classes = 6

labels_train = dataset_train.target
labels_test = dataset_test.target

#format data for the categorical cross entropy
labels_train = to_categorical(labels_train)
labels_test = to_categorical(labels_test)
```

```
def neural_model():
    model = Sequential()
    model.add(Input(shape = (vector_data.shape[1],)))
    model.add(
        Dense(
            vector_data.shape[1],
            kernel_initializer = 'normal',
            activation = 'relu'
        )
    )
    model.add(
        Dense(
            num_classes,
            kernel_initializer = 'normal',
            activation = 'softmax'
        )
    )
    model.compile(
        loss = 'categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return model

model = neural_model()
model.fit(
    vector_data_train,
    labels_train,
    validation_data = (vector_data_test, labels_test),
    epochs = 6,
    batch_size = 50
)

scores = model.evaluate(vector_data_test, labels_test, verbose = 0)
print("Two Layer Network Error: %.2f%%" % (100-scores[1]*100))
model.summary()
```

```
Two Layer Network Error: 17.64%
```

Layer (type)	**Output Shape**	**Param #**
dense_44 (Dense)	(None, 6089)	37,082,010
dense_45 (Dense)	(None, 6)	36,540

```
pred = np.argmax(model.predict(vector_data_test), axis=-1)
labels_test = np.where(labels_test == 1)[1]
```

```
print(
    f"Classification report for classifier:\n"
    f"{metrics.classification_report(labels_test, pred, digits = 4)}\n"
    f"{metrics.confusion_matrix(labels_test, pred)}\n"
)
Classification report for classifier:
              precision    recall  f1-score   support

           0     0.8903    0.8766    0.8834       389
           1     0.9138    0.8974    0.9056       390
           2     0.8054    0.9068    0.8531       397
           3     0.8015    0.8299    0.8155       394
           4     0.7482    0.6613    0.7021       310
           5     0.7288    0.6853    0.7064       251

    accuracy                         0.8236      2131
   macro avg     0.8147    0.8096    0.8110      2131
weighted avg     0.8227    0.8236    0.8220      2131

[[341  14  14  19   1   0]
 [ 11 350  12  12   4   1]
 [  4   7 360   9  14   3]
 [ 15   5  25 327  22   0]
 [  3   4  18  20 205  60]
 [  9   3  18  21  28 172]]
```

10. Large Language Models

A Large Language Model (LLM) combines many of the tools we have used in previous sections, with a new ingredient: a transformer. ChatGPT even includes this term in its name (generative pre-trained transformer.) As in Exercise 4.14, an LLM first converts text input into a set of vectors using a tokenizer (similar to what we called a vectorizer in Exercise 4.14). The LLM then feeds these vectors into several transformer layers. We will describe a transformer further below in Definition 10.1, but it is basically a variant of a neural network layer. The LLM then passes its input through a more traditional neural network. The LLM responds to a prompt by predicting a response to that prompt. An LLM trains on a large volume of questions and answers.

Let $\beta > 0$. For any $x \in \mathbb{R}^n$, define the **softmax** function $p = p^{(\beta)} \colon \mathbb{R}^n \to \mathbb{R}^n$ by

$$p_i(x) := \frac{e^{\beta x_i}}{\sum_{j=1}^n e^{\beta x_j}}, \qquad \forall\, x \in \mathbb{R}^n, \qquad \forall\, 1 \leq i \leq n.$$

Observe that $\sum_{i=1}^n p_i = 1$ and $\lim_{\beta\to\infty} p_i(x) = 1$ when $x_i = \max_{1\leq j\leq n} x_j > x_k$ for all $k \neq i$.

The quantity $1/\beta$ is called the **temperature** of the softmax function.

Below, we consider a function $f \colon \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$, i.e. f is a function from $m \times n$ real matrices to $m \times n$ real matrices. We denote the i^{th} row of and $m \times n$ matrix M as $M^{(i)}$, for each $1 \leq i \leq m$.

Definition 10.1 (Transformer). Let $A_1, \ldots, A_k, B_1, \ldots, B_k$ be $n \times n$ real matrices. A **multi-head attention layer** or **transformer** with k **attention heads** is a function $f\colon \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ defined by

$$[f(Y)]^{(i)} := \sum_{j=1}^{k} p([YA_jY^T]^{(i)})YB_j, \qquad \forall\, Y \in \mathbb{R}^{m\times n}.$$

The function f is called a transformer since it "transforms" length m sequences of n-dimensional vectors (called "tokens"). The matrices $A_1, \ldots, A_k$ are called **attention matrices**, and the matrices $B_1, \ldots, B_k$ are thought of as projection matrices, though they will in general not be actual projections matrices. Each of the k terms in the sum in Definition 10.1 is called an attention head.

A transformer is sometimes defined under the more restrictive assumption that the attention and projection matrices can be written as

$$A_j = C_jD_j^T, \qquad B_j = \begin{pmatrix} 0 & \cdots & 0 & \widetilde{B}_j & 0 & \cdots & 0 \end{pmatrix} B, \qquad \forall\, 1 \leq j \leq k,$$

where $\widetilde{B}_j, C_j, D_j$ are n by n/k real matrices (assuming n is a multiple of k) and B is a real $n \times n$ matrix.

In practice, each row of the matrix Y corresponds to a text string. The intuition for Definition 10.1 is that, once a model is trained, matrices $A_1, \ldots, A_k$ will be chosen in a way to emphasize or de-emphasize different parts of a text string in a way that extracts the important parts of the text string.

In its most basic form, a transformer is used by composing with a neural network.

If $\widetilde{f}\colon \mathbb{R}^{m\times n} \to \mathbb{R}$ is a neural network, then the function

$$\widetilde{f} \circ f$$

is what is trained to create a large language model. For a more practical example, let's consider the open source LLama-3 LLM [TLI+23] with 6.7 billion parameters, which has the 32 layer structure

$$g_{32} \circ g_{31} \circ \cdots \circ g_1,$$

where

$$g_i = (f_1 \circ r, f_2 \circ r, f_3 \circ r, \widetilde{f} \circ r), \qquad \forall\, 1 \leq i \leq 32,$$

f_j is a transformer with $n = m = 2^{12}$ for each $1 \leq j \leq 3$, $\widetilde{f}$ is a single layer (vector-valued) neural network of width $d := \lfloor (8/3) \cdot 2^{12} \rfloor$ with SiLU activation function

$$h(s) = \frac{s}{1 + e^{-s}}, \qquad \forall\, s \in \mathbb{R},$$

(more specifically, $\widetilde{f}(Y) = W_2[h(W_1Y) \odot (W_3Y)]$ where W_1 is $d \times m$, W_2 is $m \times d$, and W_3 is $d \times m$, where h is applied to each element of the matrix W_1Y and $\odot$ represents elementwise multiplication, i.e. $(A \odot B)_{ij} = A_{ij}B_{ij}$ when A, B are matrices of the same size), and the j^{th} component of r_i satisfies

$$r_{i,j}(Y) = \frac{Y_{ij}}{\sqrt{\frac{1}{n}\sum_{k=1}^{n} Y_{ik}^2}}, \qquad \forall\, Y \in \mathbb{R}^{m\times n}.$$

(The actual structure is a bit different from this but hopefully this is sufficient detail.)

Exercise 10.2. Fill in any missing details of the 6.7 billion parameter Llama-3 definition, by examining the code at

https://github.com/meta-llama/llama3/blob/main/llama/model.py.

With 6.7 billion 16 bit parameters, it takes around 14 gigabytes to store the weights from this model.

Another version of this LLM has 70 billion parameters. If these parameters each have 16 bits, it takes around 140 gigabytes to store these weights.

10.1. **Transformers for Text Classification.**

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras import layers
from tensorflow.keras.layers import Embedding, Layer, Dense, Dropout, MultiHeadAttention
from tensorflow.keras.layers import LayerNormalization, Input, GlobalAveragePooling1D
from tensorflow.keras.layers import LSTM, Bidirectional
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
import itertools
import math
from sklearn.datasets import fetch_20newsgroups

categories = [
    "comp.graphics",
    "misc.forsale",
    "rec.sport.baseball",
    "sci.space",
    "talk.politics.misc",
    "talk.religion.misc",
]

dataset_train = fetch_20newsgroups(
    remove = ("headers", "footers", "quotes"),
    subset = "train",
    categories = categories,
    shuffle = True,
    random_state = 42,
)
dataset_test = fetch_20newsgroups(
    remove = ("headers", "footers", "quotes"),
    subset = "test",
    categories = categories,
    shuffle = True,
    random_state = 42,
)
labels_train = dataset_train.target
labels_test = dataset_test.target

#format data for the categorical cross entropy
labels_train = to_categorical(labels_train)
labels_test = to_categorical(labels_test)
padding_type = 'post'
trunc_type = 'post'

tokenizer = Tokenizer()
tokenizer.fit_on_texts(dataset_train.data)
vocab_size = len(tokenizer.word_index) + 1
print("Vocab Size: ",vocab_size)
```

I got an output of 34575 for Vocab Size.

```
# truncates each token sequence to a maximum of maxlen tokens
#   without this restriction, i got out of memory (oom) errors
maxlen = 500
train_sequences = tokenizer.texts_to_sequences(dataset_train.data)
X_train = pad_sequences(
    train_sequences,
    maxlen = maxlen,
    padding = padding_type,
    truncating = trunc_type
)

test_sequences = tokenizer.texts_to_sequences(dataset_test.data)
X_test = pad_sequences(
    test_sequences,
    maxlen = maxlen,
    padding = padding_type,
    truncating = trunc_type
)
width, maxlen = X_train.shape
# transformer implementation from sklearn documentation
class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, heads, neurons):
        super(TransformerEncoder, self).__init__()
        self.att = layers.MultiHeadAttention(num_heads=heads, key_dim=embed_dim)
        self.ffn = Sequential(
            [layers.Dense(neurons, activation="relu"), layers.Dense(embed_dim),]
        )
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(0.5)
        self.dropout2 = layers.Dropout(0.5)

    def call(self, inputs, training):
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)

class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.token_emb = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=embed_dim)

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions
# output dimension of the transformer
embed_dim = 100
# number of attention heads of the transformer
heads = 3
# dimension of intermediate layer in transformer
neurons = 50

inputs = layers.Input(shape=(maxlen,))
embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size, embed_dim)
x = embedding_layer(inputs)
transformer_block = TransformerEncoder(embed_dim, heads, neurons)
x = transformer_block(x, training = True)
x = layers.GlobalAveragePooling1D()(x)
x = Dropout(0.35)(x)
outputs = layers.Dense(6, activation="sigmoid")(x)
model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer=tf.keras.optimizers.Adam(0.0003), loss='binary_crossentropy', me
model.summary()
```

This command has output
Total params: 3,851,206 (14.69 MB)

Layer (type)	**Output Shape**	**Param #**
input_layer (InputLayer)	(None, 500)	0
token_and_position_embedding (TokenAndPositionEmbedding)	(None, 500, 100)	3,507,500
transformer_encoder (TransformerEncoder)	(None, 500, 100)	343,100
global_average_pooling1d (GlobalAveragePooling1D)	(None, 100)	0
dropout_3 (Dropout)	(None, 100)	0
dense_2 (Dense)	(None, 6)	606

Table 1. Model Summary

Trainable params: 3,851,206 (14.69 MB)
Non-trainable params: 0 (0.00 B)

```
 earlystopping = EarlyStopping(
    monitor = 'loss',
    min_delta = 0.001,
    patience = 1,
    verbose = 1
)

learning_rate_reduction = ReduceLROnPlateau(
    monitor = 'loss',
    patience = 3,
    verbose = 1,
    factor = 0.2,
    min_lr = 0.00000001
)
history = model.fit(X_train,
                    labels_train,
                    epochs = 10,
                    batch_size = 50,
                    callbacks = [earlystopping]
)

 from sklearn import metrics
y_pred = np.argmax(model.predict(X_test), axis=-1)

print(
    f"Classification report for classifier:\n"
    f"{metrics.classification_report(dataset_test.target, y_pred, digits = 4)}\n"
    f"{metrics.confusion_matrix(dataset_test.target, y_pred)}\n"
)
```

```
 Classification report for classifier:
              precision    recall  f1-score   support

           0     0.8984    0.8638    0.8807       389
           1     0.9028    0.9051    0.9040       390
           2     0.9312    0.8866    0.9084       397
           3     0.7300    0.8782    0.7972       394
           4     0.7952    0.6387    0.7084       310
           5     0.6868    0.7251    0.7054       251

    accuracy                         0.8292      2131
   macro avg     0.8241    0.8163    0.8174      2131
weighted avg     0.8342    0.8292    0.8290      2131

[[336  14   6  27   2   4]
 [  8 353   4  20   2   3]
 [  4   7 352  23   5   6]
 [ 17   6   6 346  15   4]
 [  3   7   5  31 198  66]
 [  6   4   5  27  27 182]]
```

Exercise 10.3. The fetch 20 newsgroups dataset has many line breaks and other special characters in each data string. Both our transformer and two-layer neural network approaches seemed unable to get an error percentage below 17%. Try to clean the dataset (by removing these special characters) and thereby try to improve the performance of both the transformer and two-layer neural network approaches. You can also change other (hyper)parameters to try to improve the classification percentage.

11. Appendix: Notation

Let n, m be a positive integers. Let A, B be sets contained in a universal set Ω.

$\mathbb{N} = \{1, 2, \ldots\}$ denotes the set of natural numbers

$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ denotes the set of integers

$\mathbb{Q} = \{a/b \colon a, b, \in \mathbb{Z}, b \neq 0\}$ denotes the set of rational numbers

$\mathbb{R}$ denotes the set of real numbers

$\mathbb{C} = \{a + b\sqrt{-1} \colon a, b \in \mathbb{R}\}$ denotes the set of complex numbers

$\in$ means "is an element of." For example, $2 \in \mathbb{R}$ is read as "2 is an element of $\mathbb{R}$."

$\forall$ means "for all"

$\exists$ means "there exists"

$\mathbb{R}^n = \{(x_1, x_2, \ldots, x_n) \colon x_i \in \mathbb{R} \,\forall\, 1 \leq i \leq n\}$

$f \colon A \to B$ means f is a function with domain A and range B. For example,

$f \colon \mathbb{R}^2 \to \mathbb{R}$ means that f is a function with domain $\mathbb{R}^2$ and range $\mathbb{R}$

$\emptyset$ denotes the empty set

$A \subseteq B$ means $\forall\, a \in A$, we have $a \in B$, so A is contained in B

$A \smallsetminus B := \{a \in A \colon a \notin B\}$

$A^c := \Omega \smallsetminus A$, the complement of A in Ω

$A \cap B$ denotes the intersection of A and B

$A \cup B$ denotes the union of A and B

$A \Delta B := (A \smallsetminus B) \cup (B \smallsetminus A)$

$\mathbf{P}$ denotes a probability law on Ω

Let $n \geq m \geq 0$ be integers. We define

$$\binom{n}{m} := \frac{n!}{(n-m)!m!} = \frac{n(n-1)\cdots(n-m+1)}{m(m-1)\cdots(2)(1)}.$$

Let $a_1, \ldots, a_n$ be real numbers. Let n be a positive integer.

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_{n-1} + a_n.$$

$$\prod_{i=1}^{n} a_i = a_1 \cdot a_2 \cdots a_{n-1} \cdot a_n.$$

$\min(a_1, a_2)$ denotes the minimum of a_1 and a_2.

$\max(a_1, a_2)$ denotes the maximum of a_1 and a_2.

The min of a set of nonnegative real numbers is the smallest element of that set. We also define $\min(\emptyset) := \infty$.

Let $A \subseteq \mathbb{R}$.

$$\sup A \text{ denotes the supremum of } A\text{, i.e. the least upper bound of } A.$$
$$\inf A \text{ denotes the infimum of } A\text{, i.e. the greatest lower bound of } A.$$

$1_A \colon \Omega \to \{0,1\}$, denotes the indicator function of A, so that

$$1_A(\omega) = \begin{cases} 1 & \text{, if } \omega \in A \\ 0 & \text{, otherwise.} \end{cases}$$

Let $g, h \colon \mathbb{R} \to \mathbb{R}$. Let $t \in \mathbb{R}$.

$$(g * h)(t) = \int_{-\infty}^{\infty} g(x)h(t-x)dx \text{ denotes the convolution of } g \text{ and } h \text{ at } t \in \mathbb{R}$$

We let I_n denote the $n \times n$ identity matrix.

References

[Aar11] Scott Aaronson, *A linear-optical proof that the permanent is #p-hard*, Electronic Colloquium on Computational Complexity (ECCC) **18** (2011), 43.

[ACKS15] Pranjal Awasthi, Moses Charikar, Ravishankar Krishnaswamy, and Ali Kemal Sinop, *The hardness of approximation of euclidean k-means*, Preprint, arXiv:1502.03316, 2015.

[ANFSW0] Sara. Ahmadian, Ashkan. Norouzi-Fard, Ola. Svensson, and Justin. Ward, *Better guarantees for \$k\$-means and euclidean \$k\$-median by primal-dual algorithms*, SIAM Journal on Computing **0** (0), no. 0, FOCS17–97–FOCS17–156.

[BK15] Amey Bhangale and Swastik Kopparty, *The complexity of computing the minimum rank of a sign pattern matrix*, CoRR **abs/1503.04486** (2015).

[CAEMN22] Vincent Cohen-Addad, Hossein Esfandiari, Vahab Mirrokni, and Shyam Narayanan, *Improved approximations for euclidean k-means and k-median, via nested quasi-independent sets*, Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (New York, NY, USA), STOC 2022, Association for Computing Machinery, 2022, p. 1621–1628.

[CAS19] Vincent Cohen-Addad and Karthik Srikanta, *Inapproximability of Clustering in Lp-metrics*, FOCS'19 - 60th Annual IEEE Symposium on Foundations of Computer Science (Baltimore, United States), November 2019.

[CEM$^+$15] Michael B. Cohen, Sam Elder, Cameron Musco, Christopher Musco, and Madalina Persu, *Dimensionality reduction for k-means clustering and low rank approximation*, Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '15, ACM, 2015, pp. 163–172.

[Che09] Ke Chen, *On coresets for k-median and k-means clustering in metric and euclidean spaces and their applications*, SIAM Journal on Computing **39** (2009), no. 3, 923–947.

[FMS07] Dan Feldman, Morteza Monemizadeh, and Christian Sohler, *A ptas for k-means clustering based on weak coresets*, Proceedings of the Twenty-third Annual Symposium on Computational Geometry (New York, NY, USA), SCG '07, ACM, 2007, pp. 11–18.

[Gal14] François Le Gall, *Powers of tensors and fast matrix multiplication*, Preprint, arXiv:1401.7714. ISAAC 2014., 2014.

[GOR$^+$21] Fabrizio Grandoni, Rafail Ostrovsky, Yuval Rabani, Leonard J. Schulman, and Rakesh Venkat, *A refined approximation for Euclidean k-means*, preprint, arXiv:2107.07358, 2021.

[GVL13] G.H. Golub and C.F. Van Loan, *Matrix computations*, Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, 2013.

[HPM04] Sariel Har-Peled and Soham Mazumdar, *On coresets for k-means and k-median clustering*, Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '04, ACM, 2004, pp. 291–300.

[JSV04] Mark Jerrum, Alistair Sinclair, and Eric Vigoda, *A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries*, J. ACM **51** (2004), no. 4, 671–697.

[KB15] Diederik Kingma and Jimmy Ba, *Adam: A method for stochastic optimization*, International Conference on Learning Representations (ICLR) (San Diega, CA, USA), 2015.

[KMN+04] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu, *A local search approximation algorithm for k-means clustering*, Comput. Geom. **28** (2004), no. 2-3, 89–112. MR 2062789 (2005a:68210)

[Mat00] J. Matoušek, *On approximate geometric k-clustering*, Discrete Comput. Geom. **24** (2000), no. 1, 61–84. MR 1765234 (2001e:52036)

[MMR18] Konstantin Makarychev, Yury Makarychev, and Ilya P. Razenshteyn, *Performance of johnson-lindenstrauss transform for k-means and k-medians clustering*, CoRR **abs/1811.03195** (2018).

[TLI+23] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample, *Llama: Open and efficient foundation language models*, CoRR **abs/2302.13971** (2023).

USC Mathematics, Los Angeles, CA
Email address: `stevenmheilman@gmail.com`