Please provide complete and well-written solutions to the following exercises.

Due October 10, 4PM PST, to be uploaded as a single PDF or Jupyter notebook to brightspace.

Homework 4

Exercise 1 (**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 newsgroup 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 2 (**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 the Definition of spectral embedding)

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

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[200:1+n_samples, :, :]
test_images = lfw_people.images[0:200, :, :]

# 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 200 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 200 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 200 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% on the original data and around 23% on the PCA data, 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 $q = 10$ and again for $q = 50$. Did your results improve for $q = 50$?

```
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[200:1+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/200))
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.
- (Optional) For the SVC option `kernel`, try inputs other than `rbf`, and see if your results improve.

Exercise 3 (**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 $x \in \mathbf{R}^{64}$, the `SVC` function by default first embeds that vector x into $\phi(x)$, where $\phi\colon \mathbf{R}^{64} \to \mathbf{R}^n$ for n large, where ϕ satisfies $\langle \phi(x), \phi(y) \rangle = e^{-\gamma ||x-y||^2}$ for all vectors $x, y \in \mathbf{R}^{64}$, 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 $x^{(1)}, \ldots, x^{(k)} \in \mathbf{R}^n$ and let $y_1, \ldots, y_k \in \{-1, 1\}$ be given. Assume that there exists $w \in \mathbf{R}^n$ such that

$$\operatorname{sign}(\langle w, x^{(i)} \rangle) = y_i, \qquad \forall\, 1 \leq i \leq k.$$

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 $w \in \mathbf{R}^n$ and $z_1, \ldots, z_k \in \mathbf{R}$ 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)}\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?

Note: in case you have trouble accessing the MNIST dataset, you could also get it from the commands

```
from tensorflow.keras.datasets import mnist
(data_train, labels_train), (data_test, labels_test) = mnist.load_data()
```

having first run the commands

```
pip install--upgrade keras
pip install--upgrade tensorflow
```