The Mathematics of Machine Learning
Homework Set 1

Due 24 February 2022 before 13:00
via Canvas

You are allowed to work on this homework in pairs. One person per pair submits the answers via Canvas. Make sure to put both names on the submission.

1 Theory Exercises

1. [2 pt] What is the Bayes-optimal predictor f_{B} for the following loss functions?

   (a) For binary classification with $Y \in \{-1, +1\}$, the following cost-sensitive loss, which considers a false negative worse than a false positive:

   $$\ell(Y, \hat{Y}) = \begin{cases} 0 & \text{if } \hat{Y} = Y, \\ 1 & \text{if } Y = -1 \text{ and } \hat{Y} = +1, \\ 10 & \text{if } Y = +1 \text{ and } \hat{Y} = -1. \end{cases}$$

   (b) For least-squares regression, the squared error loss from the lectures: $\ell(Y, \hat{Y}) = (Y - \hat{Y})^2$.

2 Programming Exercise

The following programming exercise is to be implemented in Python, using a Jupyter notebook.

2. [8 pt]

   (a) Simulate a training set of size $N = 200$ and a test set of size 10 000 by sampling from the following binary classification distribution with $X \in \mathbb{R}^2$ and $Y \in \{-1, +1\}$:

      i. Sample a Bernoulli random variable $Z \in \{-1, +1\}$ such that $\Pr(Z = 1) = 3/4$. The interpretation is that Z represents the unobserved true class.

ii. Set $\mu_Z = Z \times \begin{pmatrix} +1 \\ -1 \end{pmatrix}$ and sample X from a normal distribution $\mathcal{N}(\mu_Z, I)$.

iii. Sample Y such that $\Pr(Y = Z) = 4/5$, so we only observe a noisy label that might differ from Z.

(b) Plot the training set in 2 dimensions, using different symbols or colors for the two classes.

(c) Similar to Figure 2.4 in the book, plot the error = average 0/1-loss both on the training set and on the test set for the K-nearest neighbour classifier as a function of $K = N, \ldots, 1$.

(d) Derive a way to compute the Bayes-optimal classifier f_{B} for the 0/1-loss, and add its error on the training set and on the test set as horizontal lines to the plot.
Hint to calculate f_{B}: Let $g(x, y, z) = \Pr(Z = z) \Pr(Y = y|Z = z)\phi(x; \mu_z, I)$ denote the joint density of X, Y and Z, where $\phi(x; \mu, \Sigma)$ is the density of a multivariate Gaussian with mean μ and covariance matrix Σ. We need to determine whether $\Pr(Y = +1|X) \geq \Pr(Y = -1|X)$, which is equivalent to

$$\sum_{z \in \{-1,+1\}} g(x, +1, z) \geq \sum_{z \in \{-1,+1\}} g(x, -1, z).$$

Programming Timesavers There are many ways to solve the above in Python, but perhaps the easiest is to implement the k-means clustering algorithm via sk-learn. The two methods I used are:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
```

which can be added to your preamble. The former command allows just out-of-the-box deployment of the k-means clustering algorithm, while the latter is a method for splitting a dataset into a training set (`x_train, y_train`) and a test set (`x_test, y_test`).

For the sake of convenience, we'll describe the latter first.

Train-test Splitting. Suppose you have your dataset of `X_samples` and `Y_samples` consisting of N examples indexed appropriately. You can make a train-test split by writing

```
X_train, X_test, Y_train, Y_test = train_test_split(
            X_samples, Y_samples, test_size = 0.9)
```

and this will randomly split your examples with 90% if the total data falling into the test set. It's as easy as that!

Deploying k-Means. You can declare a new k-means instantiation with the command

`knn = KNeighborsClassifier(n_neighbors=k).`

where `k` is the number of neighbors, and for part 2*c*) can be made iterable. The object `knn` can now be trained with the help of its method `knn.fit(X_train, y_train)`, which can be called as written here. If you'd like to see how your model is doing on your training set, you can write `knn.score(X_train, y_train)`, and likewise for the test set.

Sampling from a Bernoulli Distribution Again, there are many ways to generate a random sample, but personally, I like having distribution objects declared from a library so I can a) make use of their native methods without having to think too much and b) know for sure that I'm using the exact same distribution whenever I call from it. A nice library to import for this purpose is `scipy.stats`. In this exercise, I just want to use it's native `bernoulli` class, so I'll write

```
from scipy.stats import bernoulli,
```

and when it comes to defining an individual object from this class, we can just use

```
p = 1/4
ber_dist = bernoulli(p).
```

Now `ber_dist` is fixed with $p = 1/4$, and one can operate on it as needed throughout the code. Let's say we want to take 500 samples. It's straightforward with the `.rvs()` method:

```
nine_thousand_samples = ber_dist.rvs(9000),
```

gives us an array of 9000 individual samples from the bernoulli distribution we've defined.

Plots. Probably many of you have used `matplotlib`'s `pyplot` method before, but on the offchance you haven't, it's good to have it in your general toolkit. You can import it via

```
import matplotlib.pyplot as plt.
```

Now, if you want to make a scatter plot for your positive and negative examples, you may write

```
plt.scatter(X_positive[:,0],X_positive[:,1], color='red')
plt.scatter(X_negative[:,0],X_negative[:,1], color='blue')
plt.show(),
```

where the indices `[:,n]` indicate that the entire `n`th column is to be referenced, and the positive and negative examples have been split by some means beforehand.