SCC.323 DEEP LEARNING

WEEK 1 LAB SHEET – PYTHON REFRESHER AND PERCEPTRON/ MLP IMPLEMENTATION

EXERCISE 1: PYTHON REFRESHER

This exercise will go through some essential common commands that we use in Python. If you are already familiar with this, you can skip the exercise and move to Exercise 2 but it is recommended to at least skim to see if you are confident with the material.

For information about using Python, please refer to the "Guidance on using Python" document.

You can do this exercise either in an iPython notebook, as we did in SCC222 or you could write the commands in a .py file and execute them using terminal. In practice, people tend to use the iPython notebook approach for analysing data and the .py approach for writing and executing models.

All of the libraries that we need should already be installed in labs and the VM. However, if you want to install them on your laptop, you can do this in terminal or command prompt using pip. E.g.:

```
pip install numpy
pip install pandas
```

If you have multiple Python installations, are working in environments or with a non-standard setup, there installation of packages may be different.

To use libraries such as pandas within your code, you can use the **`import pandas`** command before any of the library commands are used.

To import libraries and rename them, you can use the **`as`** command, e.g. to import pandas and rename it as pd for your code, enter: **`import pandas as pd`**

If you want to import a sub-library, use the **`from`** command, e.g. **`from scipy import stats`**.

You may find the following references useful:

- Data structures: https://docs.python.org/3/tutorial/datastructures.html
- Dictionaries: https://docs.python.org/3/tutorial/datastructures.html#dictionaries
- Matplotlib Pyplot: https://matplotlib.org/stable/tutorials/pyplot.html
- Numpy: https://www.w3schools.com/python/numpy/default.asp
- Pandas: https://www.w3schools.com/python/pandas/pandas_getting_started.asp
- Python setup and usage: https://docs.python.org/3.11/using/index.html
- Python tutorial: https://docs.python.org/3.11/tutorial/index.html
- Python language reference: https://docs.python.org/3.11/reference/index.html
- SciPy: https://docs.scipy.org/doc/scipy/reference/main_namespace.html
- Seaborn: https://www.w3schools.com/python/numpy/numpy_random_seaborn.asp

1.1: BASIC PYTHON COMMANDS

In this section, we'll review some basic python commands for working with variables, lists and dictionaries.

1.1.1: VARIABLES AND BASIC OPERATIONS

Create two variables a and b, and assign to these the values 2 and 3 respectively. Carry out and print the following operations:

1. Sum of a and b: $a + b$
2. Difference between a and b: $a - b$
3. Product of a and b: $a \cdot b$
4. Division of a by b: $\frac{a}{b}$. Round this to 3 decimal places.
5. The remainder when b is divided by a: $b \bmod a$
6. a to the power of b: a^b
7. The exponent of a: e^a
8. The square root of a

1.1.2: WORKING WITH LISTS

Create two lists a and b with values $[2,4,6,8]$ and $[1,3,5]$ respectively.

1. What is the result if you try to add these together? Why?
2. Print the second item from list b.
3. Set c to be a copy of list a. Remove the third item from list c without writing it from scratch. Print and review the variables a and c. Is it what you expected?
4. Remove the last item from list a without explicitly specifying any indices.
5. Remove the first item from list a.
6. Add the value 8 back into list a.
7. Print the length of the list a.
8. Print the list a in reverse order.

1.1.3: DICTIONARIES

A dictionary is a collection of key and value pairs. We typically use them for storing data attributes. You can create a dictionary using braces `{` and `}`. The following syntax may be useful:

1. To create a simple dictionary with one key with one value: `d = {"key":"val"}`
   You can also do this with: `d = dict([("key","val")])`
2. To create a dictionary with one key and two values: `d = {"key":["val1","val2]}`
   or `d = dict([("key",["val1","val2"])])`
3. To create a dictionary with multiple keys and multiple values:
   `d = {"key1":["val1","val2"], "key2":["val3","val4"]}`
   or `d = dict([("key1",["val1","val2"]), ("key2",["val3","val4"])])`

Test the above commands to make sure that the syntax is correct, and note the subtle differences.

1. Create a dictionary from the following table, where the first row gives the keys.

Make	Model	Year
Austin	Mini	1959
Jaguar	E-type	1961
Clenet	Series 1	1979

Table 1

2. Print the lists of keys and values in the dictionary.
3. Return the make of the Jaguar in the dictionary.
4. Add the following item to the dictionary:

Make	Model	Year
Bugatti	Type 35	1915

Table 2

5. Delete the Austin Mini 1959 from the dictionary.
6. Use a for loop to print each row of the table from the dictionary.

You can also create nested dictionaries, which can be a more intuitive structure to work with using the syntax `d = [{"key1":"val1","key2":"val3"},{"key1":"val2","key2":"val4"}]`

1. Recreate your dictionary of Table 1 using this syntax.
2. Return the make of the Jaguar from your new dictionary.
3. Add the item from Table 2 to your dictionary.
4. Delete the Austin Mini 1959 from the dictionary.
5. Use a for loop to print each row of the table from the dictionary.

Consider the differences in syntax between the two structures. Which is cleaner and more intuitive?

1.2: NUMPY

1.2.1: ARRAYS AND MATRICES

In this section, we will create NumPy arrays and perform some fundamental operations. First, import numpy and give it the abbreviation "np".

1. Create two lists a and b with values $[2,4,6,8]$ and $[1,3,5]$ respectively, as in section 1.1.2.
2. Convert these lists into NumPy arrays.
3. Try adding the arrays together. What is the problem?
4. Remove the last element from a and try adding them again.
5. Multiply a and b together in an elementwise way. This is known as the Hadamard product.
6. Try the performing the operations of section 1.1.1 on a and b.
7. Create a column vector c with 4 random numbers, rounded to 2 decimal places.
8. Reshape c into a $2x2$ array d.
9. Print the transpose of d.

1.2.2: STATISTICS AND METRICS

In this task, we'll look at some basic statistics and metrics for analyzing data.

1. Generate an array of 50 random values from a normal distribution and store them in a variable a. The stats.norm.rvs or np.random.normal functions may be useful here.
2. Generate another array b of 50 values that is correlated with a. E.g. $b = 5 + 2a$
3. Calculate the mean, variance and standard deviation of a.
4. Calculate and print the covariance and correlation between a and b.
5. Try converting a to a pandas dataframe and printing the descriptive statistics.

1.3: GRAPHICS AND VISUALISATION

Graphics and visualisation are important for analyzing and interpreting results and models. We will use the matplotlib.pyplot, seaborn and Axes3D from mpl_toolkits.mplot3d

1. Create a scatterplot of the arrays a and b from section 1.2.2.

2. Create a scatterplot with linear regression line using regplot from the seaborn library.
3. Create a third random array c of 50 values. Examine the scatterplot and descriptive statistics.
4. Create a contour plot to examine a function of two variables:
    a. Create two variables x and y of 51 linearly spaced values between -pi and pi. You can use linspace from numpy.
    b. Create a meshgrid from x and y. What does this mean?
    c. Create a matrix z using the equation $z = \cos(x)\sin(y)$
    d. Generate a contour plot of z.
5. Try editing the equation to see the impact on the plot.
6. Create a 3D surface plot of the function z.

1.4: WORKING WITH PANDAS

Pandas is a common library for working with datasets. It is useful for storing data in tables, calculating basic statistics and visualization.

1. Import the iris dataset from github into a pandas dataframe. The dataset can be found at https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv or in the lab folder and you can use the **`read_csv`** function from pandas.
2. Print the dataframe.
3. Print the shape of the dataframe and print the column names.
4. Ensure that the measures are in numeric format using the **`to_numeric`** function.
5. Create some visualisations of the data:
    a. Create a boxplot to show the distribution of the sepal length across species. Try this with the other measures. You can also use the seaborn library for this.
    b. Create a scatter plot to show the relationship between sepal length and width
    c. Create a histogram to show the distribution of petal lengths
    d. Create a scatter matrix to show the relationship between each of the features. You will need the scatter_matrix function from pandas.plotting for this.

EXERCISE 2: IMPLEMENT PERCEPTRON AND MLP FROM SCIKIT-LEARN

In this exercise, we will learn how to use scikit-learn's built-in implementations of Perceptron and MLP for classification on a simple dataset. We will look at visualization of the decision boundaries and compare the classifiers.

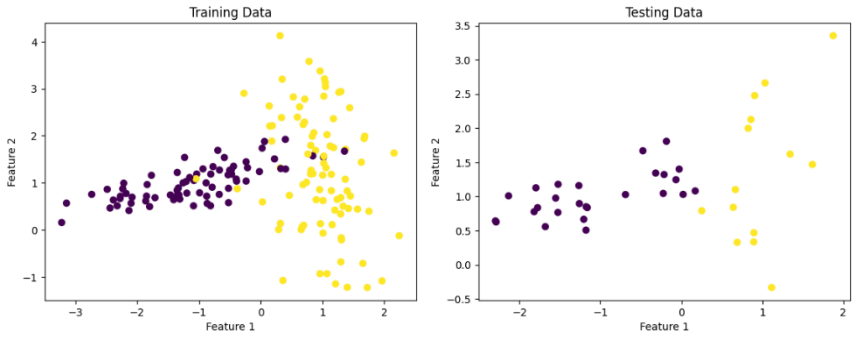
2.1: IMPORT THE LIBRARIES AND BUILD A DATASET

First, we need to import the following libraries:

1. numpy
2. matplotlib.pyplot
3. make_classification from sklearn.datasets
4. Perceptron from sklearn.linear_model
5. MLPClassifier from sklearn.neural_network
6. train_test_split from sklearn.model_selection
7. accuracy_score from sklearn.metrics

Now, build a simple 2-class dataset with two features. You can use the **`make_classification`** function for this.

Plot scatter graphs, similar to those below, of your training and testing data coloured according to the class.



2.2: TRAIN AND TEST A PERCEPTRON

1. Initialize a perceptron using the Perceptron function that you imported with 1000 iterations, learning rate of 0.01 and random state of 0.
2. Train the perceptron with your data using the **`fit`** function.
3. Use the trained perceptron on your testing data with the **`predict`** function.
4. Calculate and print the accuracy using the **`accuracy_score`** function.

2.3: TRAIN AND TEST AN MLP

1. Initialize a multilayer perceptron with 1 hidden layer of 5 neurons, 1000 iterations and random state of 0.
2. Train the MLP with your data using the **`fit`** function.
3. Use the trained MLP on your testing data with the **`predict`** function.
4. Calculate and print the accuracy using the **`accuracy_score`** function.

2.4: VISUALISE THE DECISION BOUNDARIES

You are given the following function code for visualization.

```
def plot_decision_boundary(clf, X, y, title):
    x_min, x_max = X[:,0].min() - 1, X[:,0].max() + 1
    y_min, y_max = X[:,1].min() - 1, X[:,1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200), np.linspace(y_min, y_max, 200))

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)
    plt.scatter(X[:,0], X[:,1], c=y, edgecolor='k', cmap=plt.cm.coolwarm)
    plt.title(title)
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()
```

1. Read through and understand what the plot_decision_boundary function does.
2. Use the function to plot the decision boundary of your perceptron.
3. Use the function to plot the decision boundary of your MLP.
4. Compare the accuracy of your Perceptron to your MLP. What do you notice about the decision boundaries?

2.5: EXPERIMENTATION

Try some of the following variations:

1. Try changing the learning rate and number of iterations of your perceptron and observe the effects.
2. Change the hidden layer size of your MLP to see the effects.
3. Try training and testing the Perceptron and MLP on the AND dataset. This can be defined as
   ```
   X_train = np.array([[0,0],[1,0],[0,1],[1,1]])
   y_train = np.array([0,0,0,1])
   ```
   Set the testing data to be the same as the training.
4. Try training and testing the Perceptron and MLP on the XOR dataset. This can be defined as
   ```
   X_train = np.array([[0,0],[1,0],[0,1],[1,1]])
   y_train = np.array([0,1,1,0])
   ```
   Set the testing data to be the same as the training.
5. What is the reason for the differences in success between the two examples and two models above?