COS30018- Option B- Task 2: Data processing 1

Cody Cronin-Sporys

103610020

Summary:

For this task I decided to use the code from the p1 project, specifically the load_data() function. This is where the data is loaded and processed for p1 and it includes parameters for many of the aspects we want to improve in the v0.1 code. In order to understand the code I spent a lot of time reading through it and the comments included with it. I would often compare the p1 code with the v0.1 code to see where differences were to get a better understanding. One example of this is the separation of test and training data. V0.1 loads them in separately using 2 different start and end dates, while p1 loads both in then splits them using various methods. For any parts of the code where I got stuck and didn't understand what was happening, I used the tutorial linked on the p1 github page: https://www.thepythoncode.com/article/stock-price-prediction-in-python-using-tensorflow-2-and-keras . An example of this happening was the link between the data processing and creating the model, for the p1 code I know how the model was being fit to the data until I checked the tutorial and realised it was done in a different part of the code, outside the create_model() function.

After understanding the p1 code I created a new data_processing file for v0.1 project. This file contained the load_data() function from p1. The first steps to get it working involved adjusting libraries, fixing references and moving some variables around since other files such as parameters and train do not exist. Then a large section of code needs to be removed from v0.1 that used to do the data loading and processing, which can be replaced with the function call.

Specify Start and End Dates as Inputs:

To start with we add data_start and data_end as parameters for the load_data() function. These are strings with the format yyyy-mm-dd which is the same as they originally were for v0.1

```
def load_data(data_start, data_end, tic
```

```
data_start (str): the start date for the data set (format "yyyy-mm-dd")
data_end (str): the end date for the data set (format "yyyy-mm-dd")
```

These are then used as parameters for the get_data() function which downloads the raw data from yahoo finance stock info.

```
# load it from yahoo_fin library
df = si.get_data(ticker, start_date=data_start, end_date=data_end)
```

Dealing with the NaN data issue:

NaN stands for not a number and for out dataset it becomes an issue in these lines of code.

```
# add the target column (label) by shifting by `lookup_step`
df['future'] = df['adjclose'].shift(-lookup_step)

# last `lookup_step` columns contains NaN in future column
# get them before droping NaNs
last_sequence = np.array(df[feature_columns].tail(lookup_step))
```

To create the future data we shift the regular data back by the lookup_step number, which is how many days ahead we want to predict. Because everything it being shifted back it leaves the last columns empty, so we have the NaN data issue. To solve this problem, we use this code to remove the NaN values from the data frame.

```
# drop NaNs
df.dropna(inplace=True) #
```

Splitting the Data Between Test and Train

The 2 aspects of splitting the data are the ratio between training and test data and the method by which they are split, being either random or by date. These 2 aspects are the parameters test_size and split_by_date in the load_data() function.

```
def load_data(data_start, data_end, ticker, n_steps=50, scale=True, shuffle=True, lookup_step=1, split_by_date=True,
                test_size=0.2, feature_columns=['adjclose', 'volume', 'open', 'high', 'low'], store_data=True):
```

In the code further down the split_by_date bool is used in an if statement to chose between the two different methods of data splitting, both methods use the test size variable.

Looking at the first method, where split_by_date is true, we first get a integer which acts as the proportion of the X data set based on the test_size value. E.g if test_size is 0.1 (10%) and X length is 200, then the train_samples int would be 180.

X and y are both numpy array with different sequence data stored in them. We use the train_samples int to access different index ranges of these arrays and assign them to the results variable which is returned at the end of the function. I used the link below to learn how numpy arrays work, specifically what the : means. My understanding is that numpy arrays have the format [start:end:step], in this case step doesn't matter. The important point is that if there's a semicolon with no value before or after it defaults to the 0 for start and max length for the end.

https://jakevdp.github.io/PythonDataScienceHandbook/02.02-the-basics-of-numpy-arrays.html

```
if split_by_date:
    # split the dataset into training & testing sets by date (not randomly splitting
    train_samples = int((1 - test_size) * len(X))
    result["X_train"] = X[:train_samples]
    result["y_train"] = y[:train_samples]
    result["X_test"]  = X[train_samples:]
    result["y_test"]  = y[train_samples:]
    if shuffle:
```

When split_by_date is false the data is split randomly, still using the test_size to split between training and test data. It does all this using the sklearn train_test_split function()

```
else:
    # split the dataset randomly
    result["X_train"], result["X_test"], result["y_train"], result["y_test"] = train_test_split(X, y,
                                                                                                test_size=test_size, shuffle=shuffle)
```

```
from sklearn.model_selection import train_test_split
```

Storing Data Locally

For this section a few edits to the original p1 code needed to be made since it didn't store the raw downloaded data. To store the data we first need to check if the directory to store the data exists, and if not, we create it. We also need to create a unique file name for the data we are going to be storing.

```
if not os.path.isdir("rawdata"):
    os.mkdir("rawdata")

ticker_data_filename = os.path.join("rawdata", f"{ticker}_{data_start}_{data_end}.csv")
```

Then to store the data we use the panda .to_csv function()

```
if (store_data and not os.path.isfile(ticker_data_filename)):
    df.to_csv(ticker_data_filename)
```

For loading the data we need to check that load_data is true and that the file exists with the conditions outlined in the function (so the ticker, start date and end date need to be the same). We can do this by checking the name is the same. We then use the panda .read_csv function to store the data in a data frame.

```
if(load_data and os.path.isfile(ticker_data_filename)):
    df = pd.read_csv(ticker_data_filename)
```

If the code doesn't find the file then it automatically downloads the data, even though load data was specified, I figured this would be the best solution, although have the code exit with a warning message could be an alternative. A similar thing happens with store data, if it already exists then we just skip it and move on instead of overwriting.

Full code below:

```
if not os.path.isdir("rawdata"):
    os.mkdir("rawdata")

ticker_data_filename = os.path.join("rawdata", f"{ticker}_{data_start}_{data_end}.csv")

if(load_data and os.path.isfile(ticker_data_filename)):
    df = pd.read_csv(ticker_data_filename)
else:
    # see if ticker is already a loaded stock from yahoo finance
    if isinstance(ticker, str):
        # load it from yahoo_fin library
        df = si.get_data(ticker, start_date=data_start, end_date=data_end) # TODO this is
    elif isinstance(ticker, pd.DataFrame):
        # already loaded, use it directly
        df = ticker
    else:
        raise TypeError("ticker can be either a str or a `pd.DataFrame` instances")

    if (store_data and not os.path.isfile(ticker_data_filename)):
        df.to_csv(ticker_data_filename)
```

Scaler Option and Storage

To allow the user the option of scaling the data the code uses a simple bool parameter and if statement.

```
def load_data(data_start, data_end, ticker, n_steps=50, scale=True,
```

```
scale (bool): whether to scale prices from 0 to 1, default is True
```

```
if scale:
    column_scaler = {}
    # scale the data (prices) from 0 to 1
    for column in feature_columns:
        scaler = MinMaxScaler() # TODO Fixed this line, was preprocessing.MinMaxScaler
        df[column] = scaler.fit_transform(np.expand_dims(df[column].values, axis=1))
        column_scaler[column] = scaler
```

This code works mostly the same as v0.1, scaling the wide range of prices to fit between 0 and 1. We can also see that the scaler information is stored in column_scaler[column], this is how the scaler information is saved for use later.

```
result["column_scaler"] = column_scaler
```

We can see here the column scaler is added to the result[] variable, which contains all the data to be returned at the end of the function. If we move from the data_processing file with the load_data function back to the stock_prediction file where we call the function, we can see how the scaler can be accessed.

```
data = load_data(data_start='2015-01-01'

scaler = data['column_scaler']
```