Intelligent Systems COS30018 – Task B.2

Ryan Chessum 102564760

This week I had to create a function to load and process a dataset.

1. Specify the start and end date of a dataset

```python
#my function for loading data for Intelligent Systems COS30018 Task B.2
def my_load_data(ticker, start_date, end_date, 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']):

    #check if data has been loaded before by xhecking if it's a string or dataframe
    if isinstance(ticker, str):
        #if data hasn't been loaded before
        # load it from yahoo_fin library and store it in the dataframe variable
        df = si.get_data(ticker, start_date, end_date) #set start and end date using new variables
    elif isinstance(ticker, pd.DataFrame):
        #if already loaded
        # already loaded, use it directly and make the dataframe the same as the ticker variable
        df = ticker
    else:
        raise TypeError("ticker can be either a str or a `pd.DataFrame` instances") #if neither, error
```

To specify the start and end date, I added start and end date variables as parameters to the function. At the beginning of the function, it checks weather the ticker variable is already a dataframe object or if it is a string using 'isinstance'. If it is a string the data needs to be collected from the web. The 'get_data' function used in P1 has optional variables for start and end dates. So we just add the start date and end date parameters into the function.

2. Dealing with the NaN issue

NaN means that there are blank spaces in the data. This can cause problems for us as it may affect the accuracy of our model. After researching I have found 2 solutions for this issue, the first is to remove the columns or rows that contain NaN values and the other is by filling in these spaces by adding our own data into these space. E.g adding the mean of the columns/rows before and after the NaN space.

Luckily for us, Pandas has a function that lets us drop NaN values.
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dropna.html

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

# drop NaNs
df.dropna(inplace=True)
#inplace true means that the the dropping is done on the same object instead of making and returning a new dataframe with the values dropped
```

Note that the last columns for Lookup_step contains NaNs so before we drop them, we save them in a variable as a numpy array.

3. Splitting the data into train and test data

The parameters allow us to set how we want the data to be sorted between training and test data.

```
shuffle=True, lookup_step=1, split_by_date=True,
'open', 'high', 'low']):
```

```
#split up the data into train and test data based on the parameters
if split_by_date:
    # split the dataset into training & testing sets by date (not randomly splitting)
    train_samples = int((1 - test_size) * len(X)) #int value for tranng the data based on the test size and length of X
    result["X_train"] = X[:train_samples] # training data
    result["y_train"] = y[:train_samples] # prediction training data
    result["X_test"]  = X[train_samples:] # testing data
    result["y_test"]  = y[train_samples:] # prediction testing data
    if shuffle:
        # shuffle the datasets for training (if shuffle parameter is set)
        shuffle_in_unison(result["X_train"], result["y_train"])
        shuffle_in_unison(result["X_test"], result["y_test"])
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)

# get the list of test set dates
```

In the function, the data is split within the if else statements. if split by date it determines the what data to use using the size of the X array (period of time) then add the split data to different sets. if shuffle is true, the split data is also shuffled around. If split by date isn't specified, the data is randomsed.

4. Store downloaded data on machine

To save the data we can store it in a csv file. Pandas datareader gives us a function to do this. There is a saving parameter we can use to tell the function if we want to save or not.

```
if saving:
    #save the data
    result['df'].to_csv(ticker_data_filename) #save dataframe in csv format
    if scale:
        result["column_scaler"].to_csv(scaler_data_filename) #save the scaler
```

We also save the scaler data in a separate file.

5. Scaling data

The last requirement for the function is data scaling. If scaling is set to true the function creates a dictionary to store scalers in it. Then foreach feature column in the dataframe it creates a new scaler, scales the data from the current column iteration usng the fit_transform function. Then we store the scaler n the dictionary. The scalers are stored in the return variable. We also save a copy n a csv file so we can use it for later, as shown before.

```python
#if scale variable is true, scale down data for use (like in v0.01)
if scale:
    #dictionary for storing scalers
    column_scaler = {}
    # scale the data (prices) from 0 to 1
    #loop to cycle through the columns
    for column in feature_columns:
        scaler = preprocessing.MinMaxScaler() #Assign an new scaler
        #scale the data in the dataframe
        df[column] = scaler.fit_transform(np.expand_dims(df[column].values, axis=1))
        #store the scaler in the ditionary
        column_scaler[column] = scaler

    # add the MinMaxScaler instances to the result returned
    result["column_scaler"] = column_scaler
```