COS30018- Option B- Task 5: Machine Learning 2

Cody Cronin-Sporys

103610020

Multi-Step:

The multi-step functionality allows for prediction to be made *k* days into the future. The code uses the variable LOOKUP_DAYS, which gets passed into the load_data() function as a parameter that sets the lookup_step variable.

```
LOOKUP_DAYS = 1
```

```
data = load_data(data_start=DATA_START, data_end=DATA_END, ticker=COMPANY, n_steps=PREDICTION_DAYS, split_by_date=SPLIT_BY_DATE, test_size=TEST_SIZE,
                 feature_columns=FEATURE_COLUMNS, store_data=STORE_DATA, load_data=LOAD_DATA, lookup_step=LOOKUP_DAYS)
```

Inside the load_data() function we use the lookup_step variable to set how far the dataframe column should be shifted when setting the future array of the dataframe.

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

We also use to to determine how many items are selected from the tail end when setting the last sequence values.

```
last_sequence = np.array(df[feature_columns].tail(lookup_step))
```

This last_sequence array is what we use to predict the future values and is a combination of the window size (n_steps) sequence and the lookup_step sequence.

```
# get the last sequence by appending the last `n_step` sequence with `lookup_step` sequence
# for instance, if n_steps=50 and lookup_step=10, last_sequence should be of 60 (that is 50+10) length
# this last_sequence will be used to predict future stock prices that are not available in the dataset
last_sequence = list([s[:len(feature_columns)] for s in sequences]) + list(last_sequence)
last_sequence = np.array(last_sequence).astype(np.float32)
```

Once the last_sequence array has been setup we add it to the result so it's returned from the function.

```
# add to result
result['last_sequence'] = last_sequence
```

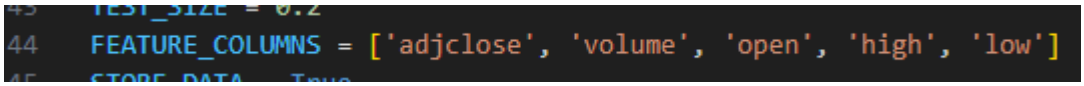
Going back to the stock_prediction code, we can find the lasts_sequence array being used in the to make a future predict based on the created model.

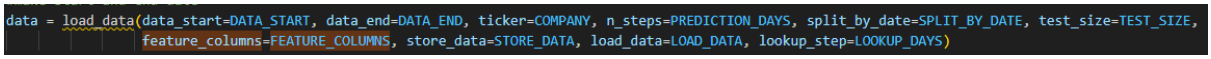
```
# retrieve the last sequence from data
last_sequence = data["last_sequence"][-PREDICTION_DAYS:]
# expand dimension
last_sequence = np.expand_dims(last_sequence, axis=0)
# get the prediction (scaled from 0 to 1)
prediction = model.predict(last_sequence)
# get the price (by inverting the scaling)
predicted_price = scaler.inverse_transform(prediction)[0][0]

print(f"Future price after {LOOKUP_DAYS} days is {predicted_price:.2f}$")
```

The first line here takes the last_sequence array and removes the n_step sequence, leaving just the future prediction days. Then the array dimensions are expanded and we make a prediction based on the model. The prediction is then scaled based on the scaler set earlier in the code. The final line prints both the *k* days value and the final prediction as a 2 decimal float.

Multi-Variate:

The simple multi-variate problem is solved in this code with the FEATURE_COLUMNS variable. This variable sets all the columns we want to use to make the predictions with, and is first passed into the load_data() function as a parameter.





The feature columns variable is used in multiple places in the load_data() function. It is first used to check if the passed columns actually exist in the downloaded dataframe. Then it's used to setup a scaler for each column.

```
# make sure that the passed feature_columns exist in the dataframe
for col in feature_columns:
    assert col in df.columns, f"'{col}' does not exist in the dataframe."
```

```
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
```

Importantly for our multi-variate problem though, the feature columns variable is used when setting up both the last_sequence and sequence_data arrays. Since the last_sequence code was explained with the multi-step problem there isn't a need to explain it again, the feature columns is just used to make sure the each of the desired columns is accounted for when adding values to the array. For the sequence_data array, the feature column is used to increment the for loop and add values to the sequences deque object (which is a list-like sequence).

```
sequence_data = []
sequences = deque(maxlen=n_steps)

for entry, target in zip(df[feature_columns + ["date"]].values, df['future'].values):
    sequences.append(entry)
    if len(sequences) == n_steps:
        sequence_data.append([np.array(sequences), target])
```

This sequence_data and sequences variables are then used to set the X and Y arrays, which are then split into X_train, y_train, X_test and y_test arrays. These test and train arrays are returned through the result variable.

```
# construct the X's and y's
X, y = [], []
for seq, target in sequence_data:
    X.append(seq)
    y.append(target)
```

```
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:
        # 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)
```

This is important, because back in the stock_prediction file, these arrays are used when fitting the model and making predictions.

```
# Now we are going to train this model with our training data
# (x_train, y_train)
model.fit(x_train, y_train, epochs=EPOCHS, batch_size=BATCH_SIZE)
```

```
predicted_prices = model.predict(x_test)
predicted_prices = scaler.inverse_transform(predicted_prices)
```

The feature column variable itself is also used as a parameter for the model_creation() function to make sure the model we generate is suitable for the desired number of columns.

```
model = create_model(PREDICTION_DAYS, len(FEATURE_COLUMNS), loss=LOSS, units=UNITS, cell=CELL, n_layers=N_LAYERS,
                     dropout=DROPOUT, optimizer=OPTIMIZER, bidirectional=BIDIRECTIONAL)
```

Summary:

In summary we use the FEATURE_COLUMNS variable to setup the last_sequence and sequence data variables in the load_data() function. These variables are then passed back and used to fit the model through the x_train and y_train arrays, predict using the model through the x_test array, and used to predict future prices through the last_sequnce array.

The multi-step and multi-variate functionality both work simultaneously, so we can make predictions *k* days into the future while also including multiple feature_columns to base our predictions on.

Much of this code was implemented earlier in the unit, and my experiments with implementing and fixing the code were included in the task 3 report. I'll include them below:

Firstly, I needed to fix an issue that had occurred due to the change in code from last weeks task. The model was not being built due to the input shape being incorrect. This was because the x_train array now stores 5 values per line instead of 1, and was fixed in the code here:

```
model.add(LSTM(units=50, return_sequences=True, input_shape=(x_train.shape[1], 5)))
```

Next the test dataset was not being created correctly due to a key issue when concatenating the test and training data. I tried changing the total dataset to fix this but it created another issue with the scaler not having a transform function because it is a dict. This was fixed by making adjusting the scaler to use the adjclose feature column.

```
total_dataset = data['df']
total_dataset = total_dataset[PRICE_VALUE]
```

```
scaler = data['column_scaler']
scaler = scaler['adjclose']
```

This then caused another issue. Due to the fix with the x_train data and the model shape, there was now a mismatch between test data and model expected shape. The model expected 5 feature columns but the current test data process only uses close. To fix this I removed total_dataset[PRICE_VALUE] line from the code. This caused other issues so I looked through the p1 code and realised that most of the code in v0.1 was unnecessary, so I commented it out and replaced it with the x_test and y_test datasets which fixed all the problems after some scaling and inverse scaling