Udacity ML Engineer Nanodegree Capstone Project Report

Predicting Bitcoin Movement Using MLP

Wah Chi Shing, Anthony

Background

As mentioned in the proposal, Bitcoin has again been brought under spotlight thanks to its ATH (all-time high) being refreshed everyday. This project aims at predicting movement in short-term bitcoin price by taking advantage of Machine Learning/Deep Learning algorithm and the computation power provided by AWS Sagemaker.

Figure 1: Bitcoin Price Surge and rose above $40000

Define Problems and Targets

There are different ways to predict an asset price, such as predicting the actual price, expected forward % change, volatility, price range ... etc. Due to the fact that number of investors in Bitcoin and the pool size (USDT used for trading BTC) change over time, the expected actual price may lose its reference value if either of them changes. Therefore, first of all, we will target on the expected forward % change of Bitcoin over a fixed time interval in this model (e.g. forward 1-minute % change). For instance, we will start with 1-minute forward price changes and 5, 10 subsequently if the result from 1-minute is not satisfactory.

The second problem is to choose target features to be used for predicting forward % change. This looks very much like a time-series problem and it seems trivial to use RNN to solve this problem. However, we don't want the price data to be the only input for the problem (or else it will be too simple that anyone with a RNN model can do it). We are trying to include some other proprietary information computed on our own analysis (e.g. moving average price, average volume traded ... etc). Therefore, we will compute a list of potential features to be included, and pick the most relevant ones to be used. More details can be found in `Ext_features_formula.ipynb`.

Rolling standardised price

`15m/60m_z_price`

Giving a fixed time interval, this feature is computed by taking a moving average and moving standard deviation of the price, and calculating the standard score of the last price of that rolling period.

```
In [8]: helper.plot_feature(['15m_z_price', '60m_z_price'], n=500)
```

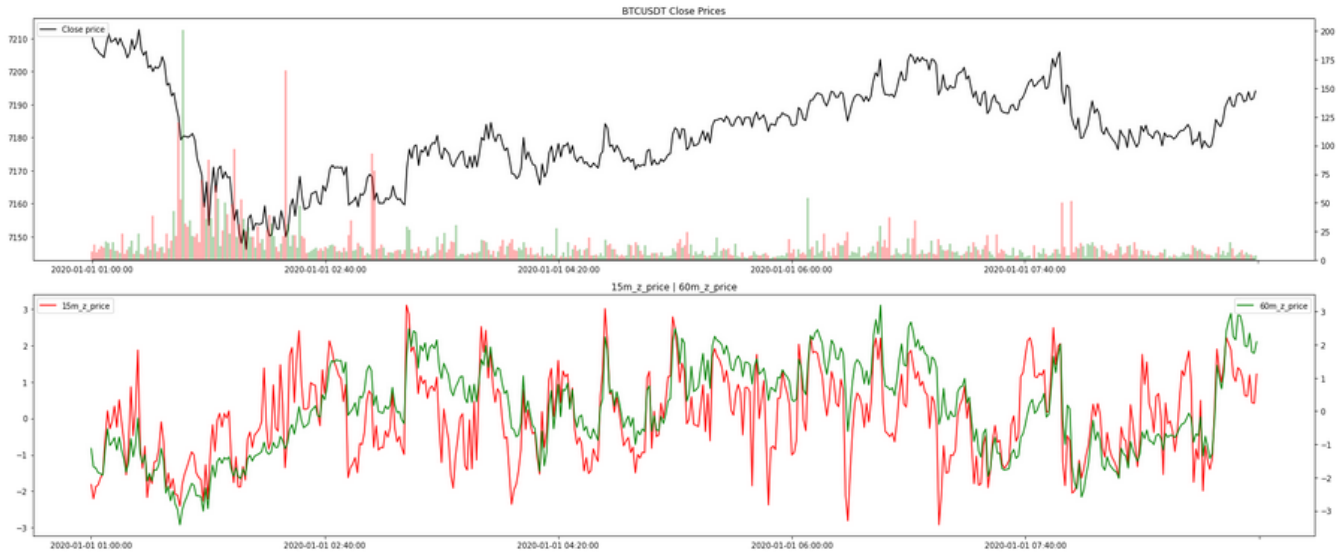

Figure 2: One feature exmaple in Ext_features_formula.ipynb

All in all, to construct the problem, we will need to:

1) Pick some target time intervals for constructing forward % change to predict, and;
2) Model a pool of features so that we can train a model based on the feature-expected-change relationship, and ;
3) Select the most relevant features based on their correlative matrix, feature formula, . . . etc

After that, we will have to build an MLP for predicting the forward % change. Note, we will try to first predict the NUMERICAL value of the forward % change directly using the model, so we will try to employ a regressor algorithm instead of a classification algorithm. (i.e., `regressor` instead of `binary_classifier` for kwargs `predictor_type` in LinearLearner, and no Sigmoid function is needed for MLP)

Data Source and Analysis

Bitcoin are traded in a lot of different crytocurrency exchanges, so its price/volume data may vary between different venues (but the difference shouldn't be too large, see here for more). Here we will use historical Bitcoin price data from Binance as it is one of the largest crytocurrency exchanges in the globe. We will get the Bitcoin 1-minute open-high-low-close-volume data from its publicly available API. `binance_data_calling.py` is a sample script for calling candlestick data from Binance with the help of a third-party package `python-binance`. Note that one needs to have an active account in Binance and have activated the API key so that API service can be used.

date_time	open	high	low	close	volume
2020-11-19 00:00:00	17873.89	17880.12	17828.51	17870.0	235.474692
2020-11-19 00:01:00	17870.0	17939.13	17870.0	17936.43	149.612749
2020-11-19 00:02:00	17936.44	17960.0	17927.63	17941.68	201.725584
2020-11-19 00:03:00	17941.68	17978.78	17937.82	17978.78	154.594181
2020-11-19 00:04:00	17975.7	17982.76	17958.0	17969.09	163.60663
2020-11-19 00:05:00	17969.08	17985.58	17953.53	17958.5	125.26772
2020-11-19 00:06:00	17958.5	17965.13	17925.01	17963.58	113.729178
2020-11-19 00:07:00	17963.59	18000.0	17962.32	17988.96	209.569132

Figure 3: Binance data sample

`1_data_preparation_and_exploration.ipynb` has demonstrated the structure of the data retrieved from Binance. All data are stored in `input_data` and sorted by different dates. All minute data in 2020 are included in this project. We should expect a time-series plot as below:

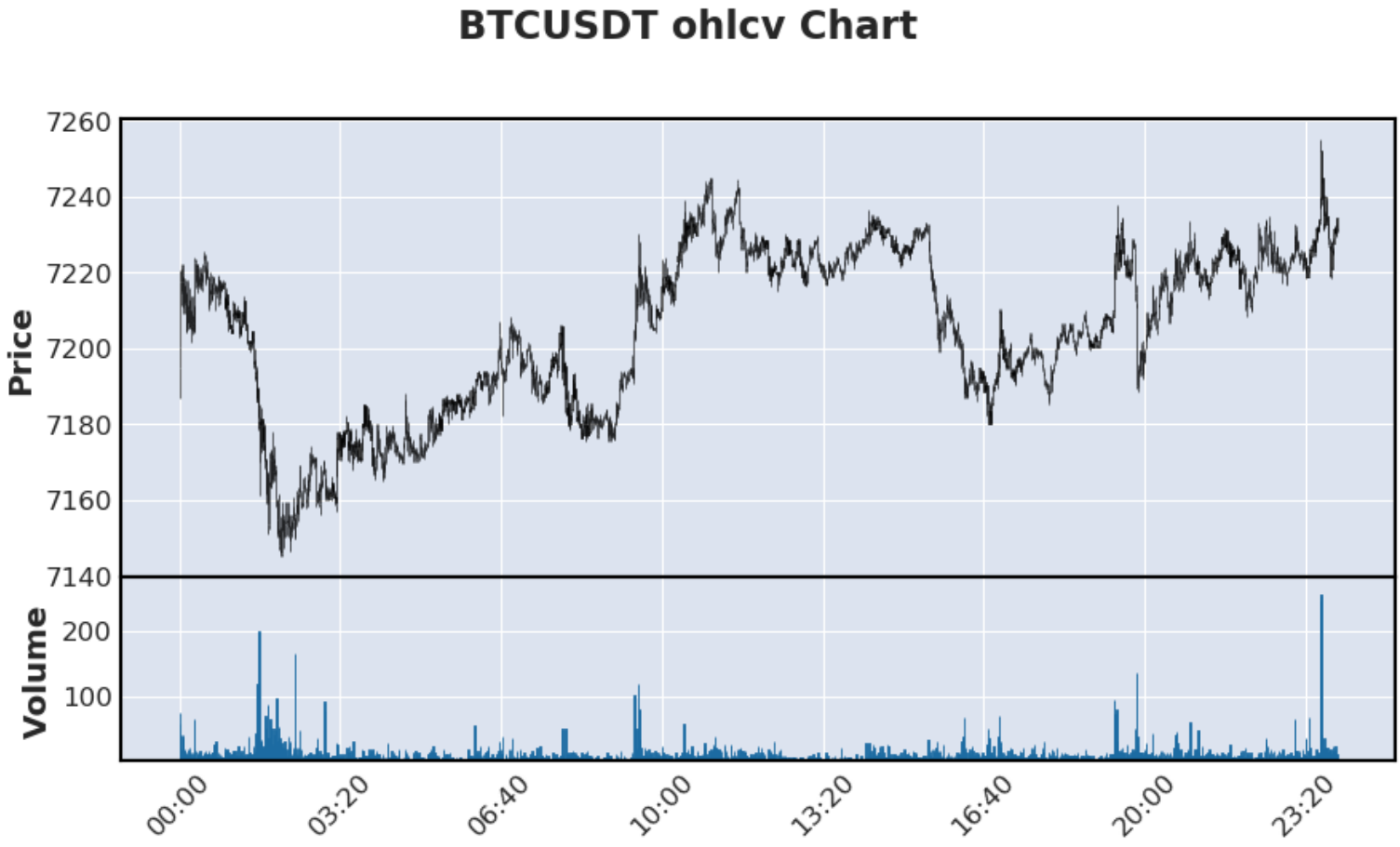

Figure 4: Bitcoin 2020 Price Chart

Also, since we are tryinbg to predict BTC's 1-minute price movement, we can plot the distribution of price changes. From the graph below, we can approximate that the price changes has a normal distribution with mean around 0.

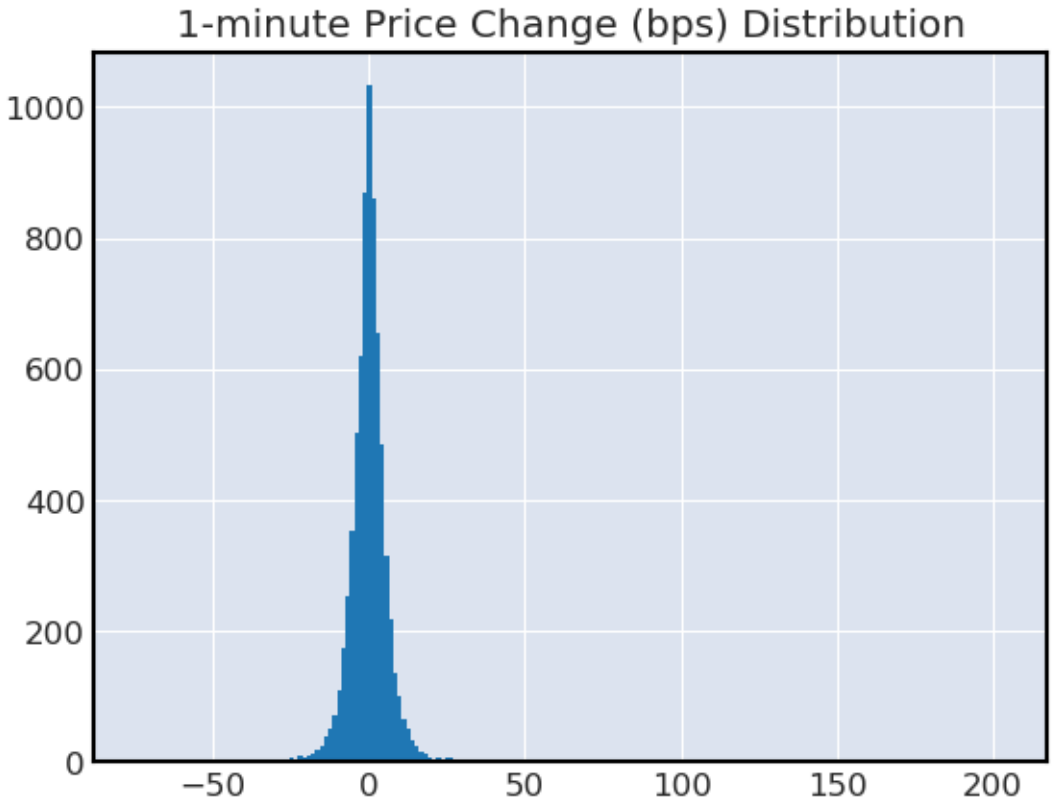

Figure 5: Bitcoin Price Changes Distribution

Benchmark Model

As mentioned in the proposal, a less complicated model as a benchmark for comparison as it can be much more easily computed. We would like to use a Moving Average convergence divergence (MACD) model, which predicts whether the price of Bitcoin will go up/down after a fixed time interval, to compare with the predictive power of our model. The formula of MACD is:

$$\text{MACD} = \text{EMA}_{12}(\text{price}) - \text{EMA}_{26}(\text{price})$$

$$\text{Signal} = \text{EMA}_{9}(\text{MACD})$$

$$\text{Divergence}_t = \text{MACD}_t - \text{Signal}_t$$

$$\text{Prediction} = \begin{cases} \text{Up} & \text{if Divergence}_t > 0 \text{ and Divergence}_{t-1} < 0 \\ \text{Down} & \text{if Divergence}_t < 0 \text{ and Divergence}_{t-1} > 0 \end{cases}$$

For comparison simplicity, we will simply take the following criterion for our model:

$$\text{Binary Prediction} = \begin{cases} \text{Up} & \text{if Numerical Prediction}_t > 0 \\ \text{Down} & \text{if Numerical Prediction}_t < 0 \end{cases}$$

`2_benchmark_model_computation.ipynb` stores the code for computing this benchmark model and the evaluation result. The graph below shows how MACD will change with time and BTC price.

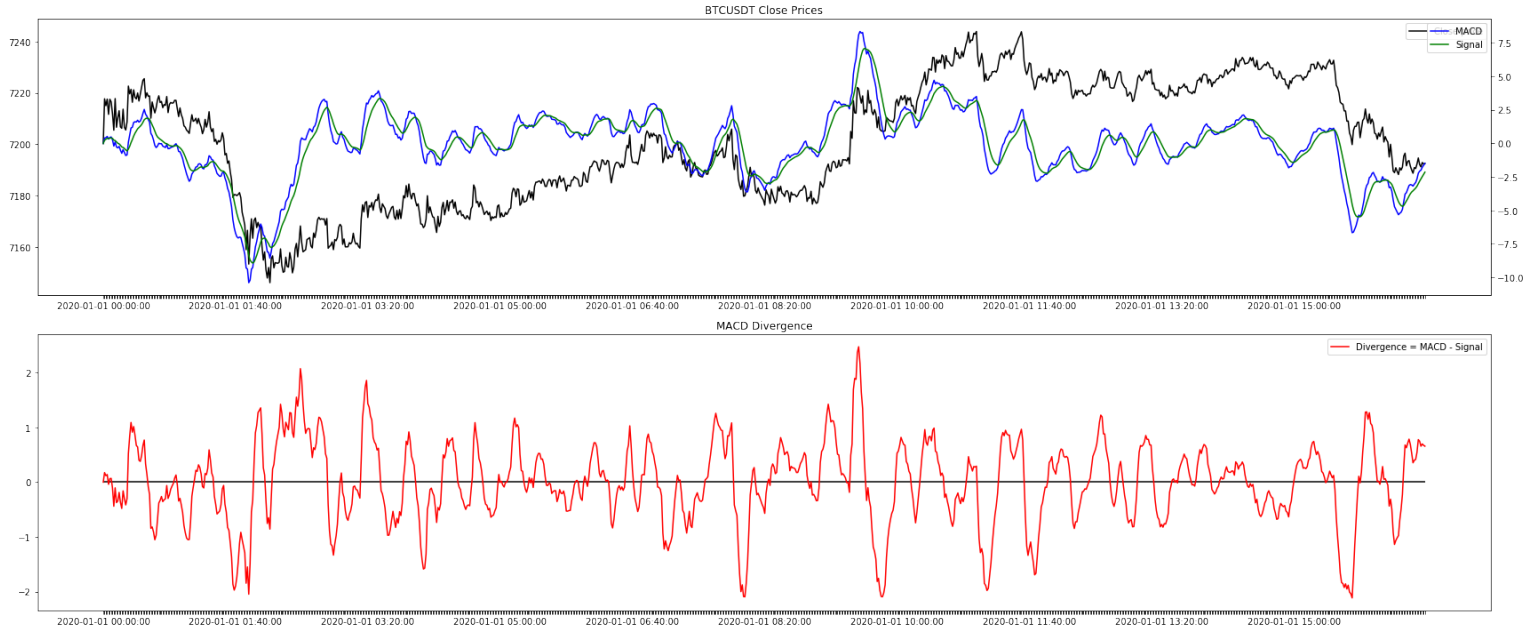

Figure 6: MACD as a benchmark

Using the prediction criteria defined above, we have the following result:

$$\text{Prediction frequency} = 8.12$$

$$\text{Prediction accuracy} = 54.12$$

`8.12%` of prediction freqency means that on average 8 signals will be yielded per 100 minute, and out of these `8.12%` predictions, `54.12%` of them will be able to indicate a correct price movement direction. Let us keep these numbers for comparison after our target model is completed.

Model Implementataion & Training Results

Features processing

First of all, we need to compute the features to be selected as inputs. The detailed process of doing so can be found in `3_target_features_processing.ipynb`. Basically, a pool of different features, including technical indicators, volume indicators, momentum indicators are computed and analyzed based on their correlation.

60m_z_volume	adx	close_to_high	close_to_low	close_to_open	fso	log_volume	macd	obv	rsi	sso	volume_chg
-0.634879	35.520157	-3.244385	1.692353	-3.230525	-33.500456	1.998947	-0.552226	14.729818	-6.830672	-17.061683	-0.593879
-0.097864	37.040723	-3.911234	3.567160	-3.911234	-32.481254	2.612076	-0.789266	1.102508	-12.943865	-28.997854	0.846199
-0.722088	38.071884	-1.845172	1.609879	-1.650975	-35.957737	1.942744	-0.921642	-5.875362	-13.198074	-33.979816	-0.487950
-0.406542	39.261958	-4.397476	0.000000	-1.137888	-43.865031	2.295279	-1.020602	-15.802565	-21.857694	-37.434674	0.422670
-0.149173	40.955101	-1.720731	1.318703	-0.707792	-43.219129	2.491547	-1.049918	-27.882509	-15.175392	-41.013966	0.216853
...	...	...	...	...	...	...	...	...	...	...	...
0.234962	52.076000	-9.940741	16.014912	-7.109959	16.839288	4.604367	5.115168	-282819.805420	19.544072	30.425002	0.951167
-0.633874	51.149409	-3.711356	10.324624	-3.329214	8.171975	3.848640	1.391930	-282866.734625	14.751922	18.546363	-0.530331
-0.535997	49.379733	-4.637206	8.115242	-4.637206	-2.113333	3.952278	-2.185655	-282918.788435	9.918063	7.632643	0.109199
-0.151573	46.370216	-11.564266	13.641102	-9.356719	-23.828571	4.312129	-6.289268	-282993.387613	4.839076	-5.923310	0.433117
-0.838498	43.847343	-10.429246	2.781087	-10.380961	-44.664286	3.442837	-10.747128	-283024.663169	-8.079789	-23.535397	-0.580752

Figure 7: Features computed

For instance, `60m_z_volume` means taking the z-score of current minute volume based on the mean-variance statistic of volume in the past 60 minutes. More details about how different features are comoputed can be found in `Ext_features_formula.ipynb`.

Then, we can look into how these features correlate with each other, by calling class method `pd.DataFrame.corr()` and visulize it by `seaborn.heatmap()` so that we have a nice colormap to understand the numerical scale of correlation between different features.

```
In [10]: import seaborn as sns

         sns.set(rc={'figure.figsize':(20, 10)})
         sns.heatmap(corr_df, annot=True, annot_kws={"fontsize": 8}, cmap='OrRd')

Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7fdbac372668>
```

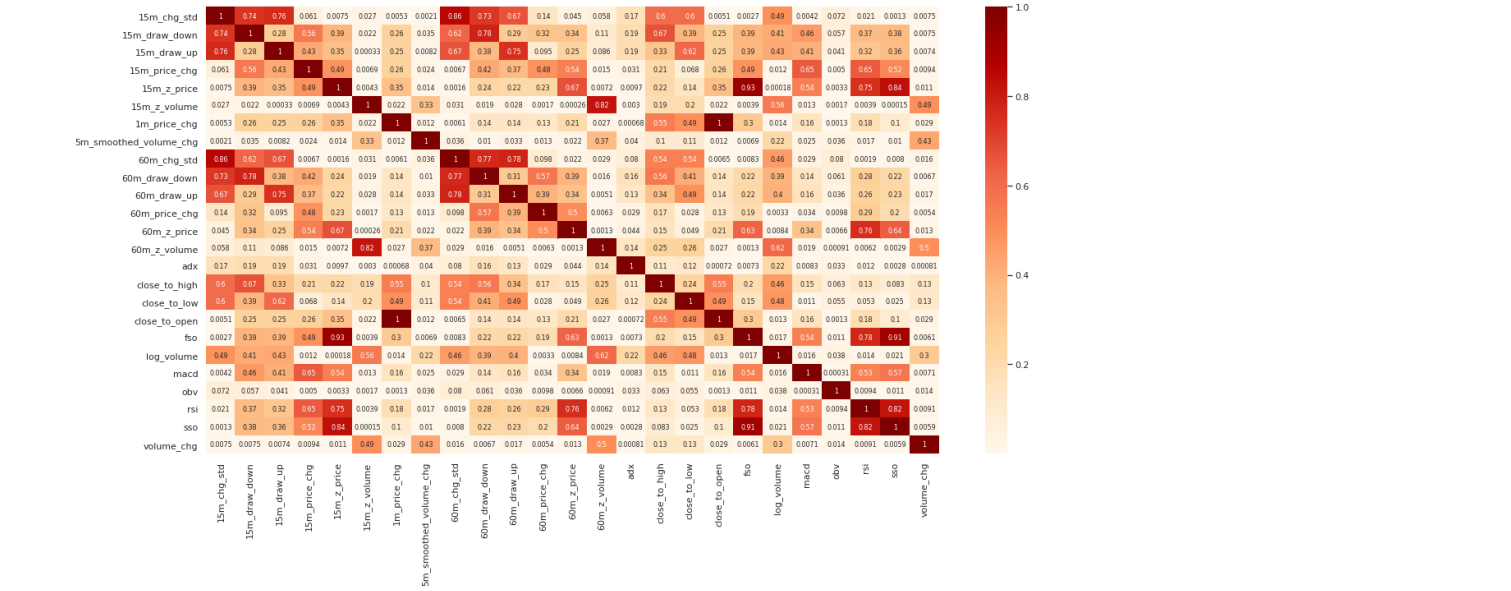

Figure 8: Intra-features Correlation

From the chart, we can find out several pairs of features which have high correlation, for example:

- Generally all `15m` and `60m` features have higher correlation as they only have a difference in moving window period
- `15m_z_price` and `fso` due to their similarity in construction formula ... etc

Data Processing

After analyzing the features' correlation, we move to the data processing part. As mentioned before, we have to pick a list of features as the model inputs in the training part. Here we have selected the following features:

```
target_features_list = [
    '60m_chg_std',
    '60m_z_price',
    '60m_z_volume',
    '60m_draw_up',
    '60m_draw_down',
    '5m_smoothed_volume_chg',
    'log_volume',
    'close_to_high',
    'close_to_low',
    'close_to_open',
    'adx',
    'macd',
    'fso',
]
```

`features_helper.py` contains a wrapper class that help us compute specified features without having to replciate the codes in the notebook again. Similar helper functions/classes are in other scripts such as `data_reader.py`, `data_processing.py`.

Therefore, we only need to import the corresponding module and pick the right functions to run. After all, we want a pandas DataFrame object with columns [y, x0, x1, x2 ... xn] for train-test set splitting. The following block summarized the computaion codes.

```
from data_reader import *
from features_helper import FeaturesHelper

df = read_all_csvs()
helper = FeaturesHelper(df)
# target_features_list defined above
helper.run_features_list(target_features_list, log=True)
df = helper.get_result(bool_dropna=True)
df = preprocess_data(df, input_period=1)  # input_period means the forward price changes target period
```

And then we should get a DataFrame like this:

	date_time	1m_fwd_chg	60m_chg_std	60m_z_price	60m_z_volume	60m_draw_up	60m_draw_down	5m_smoothed_volume_chg	log_volume	close_to_high	close_to_low	close_to_open	adx	macd	fso
0	2020-01-01 01:00:00	-4.077602	0.000704	-1.123762	-0.634879	13.749580	-21.451446	0.442304	1.998947	-3.244385	1.692353	-3.230525	35.520157	-0.552226	-33.500456
1	2020-01-01 01:01:00	-0.707628	0.000631	-1.679798	-0.097864	7.720485	-25.520301	0.033968	2.612076	-3.911234	3.567160	-3.911234	37.040723	-0.789266	-32.481254
2	2020-01-01 01:02:00	-1.609620	0.000630	-1.712799	-0.722088	7.012311	-26.226123	0.008016	1.942744	-1.845172	1.609879	-1.650975	38.071884	-0.921642	-35.957737
3	2020-01-01 01:03:00	-0.707792	0.000628	-1.856050	-0.406542	5.401562	-27.831522	0.043619	2.295279	-4.397476	0.000000	-1.137888	39.261958	-1.020602	-43.865031
4	2020-01-01 01:04:00	-1.054823	0.000620	-1.882046	-0.149173	4.693388	-28.537344	0.080779	2.491547	-1.720731	1.318703	-0.707792	40.955101	-1.049918	-43.219129

Figure 9: Processed DataFrame

As for the train-test splitting part, note that since we are splitting time-series data, we can't not shuffle and rely on `train_test_split` function from `sklearn` library. We need to split it manually.

```
# Splitting train and test dataset
ratio = 0.6
training_df = df.loc[df.index[:int(len(df) * ratio)], :].reset_index(drop=True)
testing_df = df.loc[df.index[int(len(df) * ratio):], :].reset_index(drop=True)
print(f'lenght of training_df = {len(training_df)}; date_time from {training_df.loc[0, "date_time"]} to {training
```

```
print(f'lenght of testing_df  = {len(testing_df)}; date_time from {testing_df.loc[0, "date_time"]} to {testing_df

# Droping date_time for saving csv
training_df.drop('date_time', axis=1, inplace=True)
testing_df.drop('date_time', axis=1, inplace=True)
```

Data Uploading

To train our model, we need to upload the train and test DataFrame built to our AWS S3 buckets. We need to first save our data to a local directory before doing so.

```
data_dir = 'processed_data'
prefix = 'sagemaker/capstone_capstone'
force_update = True

# Save csv to be uploaded
data_path = os.path.join(data_dir, 'train.csv')
if force_update or not os.path.exists(data_path):
    save_df(input_df=training_df, save_path=data_path)
    print('Saved at', data_path)
else:
    print(data_path, 'exists already')

# Upload to S3, if object doesn't exist
if force_update or os.path.join(prefix, 'train.csv') not in [obj.key for obj in boto3.resource('s3').Bucket(bucke
    s3_input_data_path = sagemaker_session.upload_data(path=data_dir, bucket=bucket, key_prefix=prefix)
    print(s3_input_data_path)
else:
    print(os.path.join(prefix, 'train.csv'), 'exists already')
    s3_input_data_path = os.path.join('s3://sagemaker-us-east-2-151738229005', prefix, 'train.csv')
```

Then we are ready to move on to model building and training part.

Target Model Building

There are 3 scripts in `source` to be taken as input by the Sagemaker PyTorch Estimator, namely `model.py`, `train.py` and `predict.py`. We have defined our model, using MLP with self defined number of hidden layers and dimensions (as class inputs), in the `model.py`.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class DynamicNet(nn.Module):
    def __init__(self, input_dim, hidden_dim_list, output_dim):
        ''' Defines layers of a neural network.

        param input_dim:
            - Number of input features
        param hidden_dim_list:
            - List of hidden layer dimensions
        param output_dim:
            - Number of output

        '''
        assert len(hidden_dim_list) > 0 and all([isinstance(i, int) for i in hidden_dim_list])
```

```
        super(DynamicNet, self).__init__()
        full_dim_list = [input_dim] + hidden_dim_list + [output_dim]
        self.full_dim_list = full_dim_list
        self.fc_dict = {}
        for i in range(len(full_dim_list) - 1):
            fc = nn.Linear(full_dim_list[i], full_dim_list[i + 1])
            setattr(self, f'fc{i + 1}', fc)
            self.fc_dict[i + 1] = f'fc{i + 1}'
        self.dropout = nn.Dropout(0.15)

    def forward(self, x):
        ''' Feedforward behavior of the net

        param x:
            - A batch of input features
        return:
            - A single, sigmoid activated value

        '''
        for i in range(len(self.fc_dict)):
            fc = getattr(self, self.fc_dict[i + 1])
            x = fc(x)
            if i + 1 != len(self.fc_dict):
                x = self.dropout(x)
        return x
```

The `DynamicNet` class enables one to define the number of hidden layer and dimensions every time when one constructs an instance by the `hidden_dim_list` parameter. For example, an input of `4_8_6` will give me 3 hidden layer of dimensions 4, 8 and 8 respectively.

Target Model Training

We will import PyTorch model from sagemaker modules, constuct an estimator instance and train it like what we have done in the previous modules. The full details of model training and prediction are available in `4_target_model_training_and_evaluation.ipynb`.

```
from sagemaker.pytorch import PyTorch

output_path = f's3://{bucket}/{prefix}'

estimator = PyTorch(entry_point='train.py',
                    source_dir='source',
                    role=role,
                    framework_version='1.0',
                    py_version='py3',
                    train_instance_count=1,
                    train_instance_type='ml.c4.xlarge',
                    output_path=output_path,
                    sagemaker_session=sagemaker_session,
                    hyperparameters={'input_dim': len(target_features_list),
                                     'hidden_dim_list': "8_6",
                                     'output_dim': 1,
                                     'epochs': 200})
estimator.fit({'train': s3_input_data_path})
```

```
2021-01-12 08:41:20 Training - Training image download completed. Training in progress.Epoch: 1  , Loss: 153.6959654775 , used 5.467s
Epoch: 2  , Loss: 153.3265464142 , used 5.334s
Epoch: 3  , Loss: 153.3139246123 , used 5.355s
Epoch: 4  , Loss: 153.2973185577 , used 5.336s
Epoch: 5  , Loss: 153.30783859   , used 5.406s
```

Figure 10: Trainig started successfully

As indicated in the hyperparameters input, we will loop over the dataset 200 times to try reducing training loss. However, the training result is not satisfactory.

```
Epoch: 191, Loss: 153.294640031  , used 5.322s
Epoch: 192, Loss: 153.2869373129 , used 5.349s
Epoch: 193, Loss: 153.2827498089 , used 5.405s
Epoch: 194, Loss: 153.300528503  , used 5.416s
Epoch: 195, Loss: 153.2854396495 , used 5.425s
Epoch: 196, Loss: 153.2784834012 , used 5.411s
Epoch: 197, Loss: 153.2893415598 , used 5.322s
Epoch: 198, Loss: 153.3040071529 , used 5.353s
Epoch: 199, Loss: 153.2838518875 , used 5.572s
Epoch: 200, Loss: 153.2909421463 , used 5.698s
Saving the model.
2021-01-12 08:59:10,573 sagemaker-containers INFO     Reporting training SUCCESS

2021-01-12 08:59:34 Uploading - Uploading generated training model
2021-01-12 08:59:34 Completed - Training job completed
ProfilerReport-1610440674: NoIssuesFound
Training seconds: 1125
Billable seconds: 1125
CPU times: user 3.18 s, sys: 144 ms, total: 3.33 s
Wall time: 21min 57s
```

Figure 11: Training Logs

As you can see from the progression in training loss, after 200 loops in dataset the model seems to have learned NOTHING from the input features. There may be numerous reasons causing this result but before that let us continue with evaluating the result first.

Target Model Evaluation

As mentioned earlier, let's try plotting the predicted values against the actual values (forward price changes) for both training and testing dataset. The first graph will be a normal plot including all values, and the second plot is obtained by 1) ranking all pair of points by ascending order in actual values, and 2) taking average of both predicted and actual values per 2000 points, such that we can have a rough idea of how these pairs are distributed.

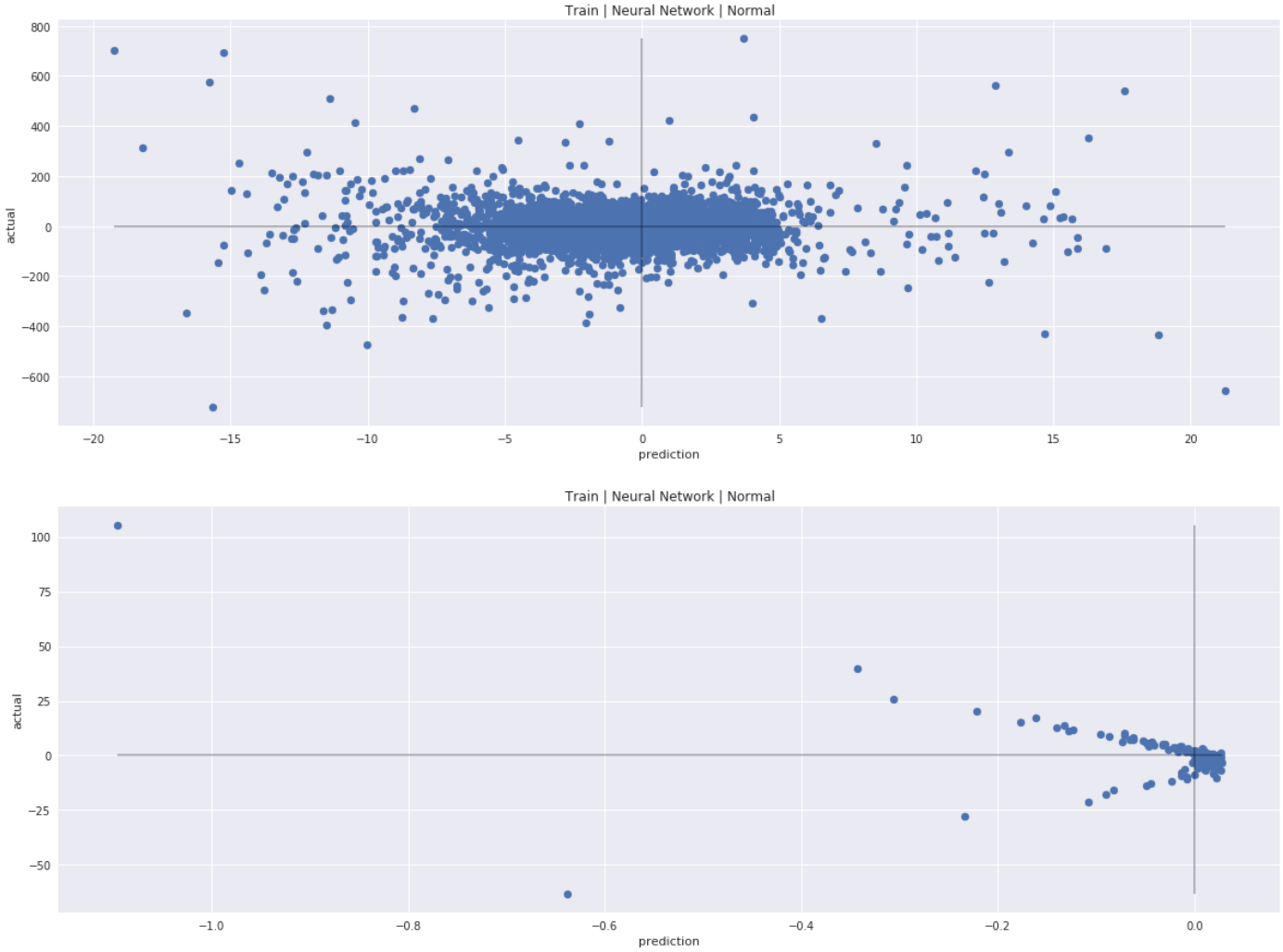

Figure 12: Training dataset

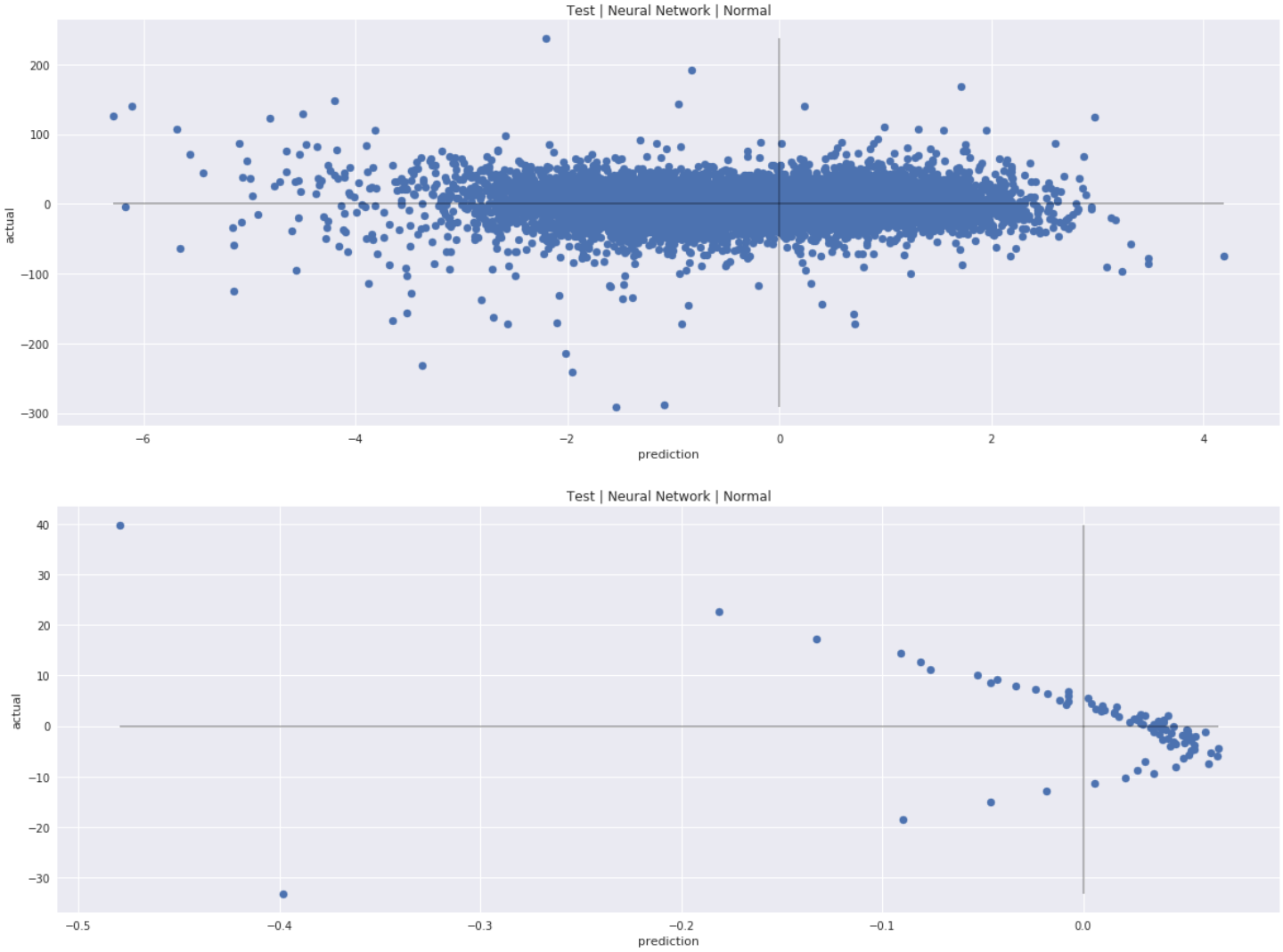

Figure 13: Testing dataset

It is trivial that the scatter shows that the neural network failed to capture the relationship between the features and the target output. We can use the following fuction to compute some metrics for further evaluation.

```
def evaluate_one_set_result(pred_array, label_array):
    df = pd.DataFrame({'label': label_array, 'pred': pred_array})
    signed_df = np.sign(df[['label', 'pred']])
    correct_sign_pc = signed_df.loc[signed_df['label'] == signed_df['pred'], :].shape[0] / signed_df.shape[0]
    corr = df.corr().iloc[0, 1]
    mse = (df['pred'] - df['label']).pow(2).sum() / df.shape[0]
    return {
        'count': df.shape[0],
        'sign_accuracy': correct_sign_pc,
        'correlation': corr,
        'mse': mse
    }

def evalute_result(train_pred_array, train_label_array, test_pred_array, test_label_array):
    res_dict = {}
    res_dict.update({f'train_{k}': v for k, v in evaluate_one_set_result(train_pred_array, train_label_array).ite
    res_dict.update({f'test_{k}': v for k, v in evaluate_one_set_result(test_pred_array, test_label_array).items()
    return res_dict
```

train_count	313194.000000
train_sign_accuracy	0.485210
train_correlation	-0.018706
train_mse	154.385640
test_count	208796.000000
test_sign_accuracy	0.483122
test_correlation	-0.022814
test_mse	73.277936

Figure 14: Result Metrics

It shows that the network failed to predict even the direction forward price changes in both testing and train set, not to mention its high average squared error. Also, the negative correlation also shows that the model is invalid for solving this problem. Still, we may try out different alternatives in approaching this problem. For instance, in `5_model_improvement_alternatives`, we have tried adding more hidden neuron layers, using 10-minutes forward price changes, or even use a LinearLearner algorithm to predict the target variable. Unfortunately, no surprising results have been observed.

	original	more_layers	less_features	more_features	5_minutes	10_minutes	linear_learner
train_count	313194.000000	313194.000000	313194.000000	313194.000000	313191.000000	313188.000000	313188.000000
train_sign_accuracy	0.485210	0.485025	0.485316	0.486162	0.470492	0.464379	0.518749
train_correlation	-0.018706	-0.018693	-0.011327	-0.012899	-0.031347	-0.031552	0.040273
train_mse	154.385640	154.259726	153.872293	153.860969	749.321619	1445.313538	1405.242407
test_count	208796.000000	208796.000000	208796.000000	208796.000000	208795.000000	208793.000000	208793.000000
test_sign_accuracy	0.483122	0.483414	0.485028	0.485982	0.465021	0.458382	0.516957
test_correlation	-0.022814	-0.022337	-0.014262	-0.020537	-0.044105	-0.050829	0.059406
test_mse	73.277936	73.207123	73.045933	72.961902	357.285031	692.100388	660.101045

Figure 15: Results of all alternatives

In the summary section, we will try to conclude some reasons for failure of this model.

Benchmark Comparison

Let's recall what results we have got in the benchmark model:

> `8.12%` of prediction freqency means that on average 8 signals will be yielded per 100 minute, and out of these `8.12%` predictions, `54.12%` of them will be able to indicate a correct price movement direction. Let us keep these numbers for comparison after our target model is completed.

Even the frequency of prediction is much less that the neural network model, our benchmark model can still successfully predict more than half of the cases in terms of the forward price changes directions. LinearLearner seems to give better result with higher accuracy in predicting the sign, the much larger MSE suggests that it is also not the optimal choice for solving this problem.

Project Conclusions

It seems that we can easily make a conclusion that "Deep learning is not applicable to Bitcoin price movement prediction". However, there are quite a number of points that this project has not specified:

1) Bitcoin price may behave differently during different time periods, e.g., day/night, first/last 5 mins of the hour
2) Bitcoin price may behave differently with a different volume profile, e.g., 200 coins traded in last 5 minutes vs 2000 coins traded in last 5 minutes
3) The model above only employed features generated from the open, high, low, close, volume data of Bitcoin price

Without specifying these factors, predicting price movements will be much more difficult for neural network since the hidden patterns among data are changing over time, and thus impossible to be optimised. This explained the possibility why the above model simply didn't work.

Also, from a financial market point of view, if we can predict the price movements of Bitcoin simply from the historical price information, it indicates that BTCUSDT market is highly inefficient, which is intuitively impossible for a cryptocurrency market where trillions of dollars are flowing on a daily basis.

Therefore, although the above model is invalid for predicting BTCUSDT price movements, it still serves as a very good starting point of how one can approach the most commonly faced problem in financial industry: price movement prediction. Hope that this project can provide a good reference for those who just started your journey in deep learning and financial industry.