Expertise
and insight
for the future

Won Seob Seo

Using market and news data to predict price movement of stocks

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

30 August 2018

Author Title Number of Pages Date	Won Seob Seo Using market and news data to predict price movement of stocks 30 pages + 0 appendices 20 February 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Supervising teacher

Until recently, investing use to be the domain of human experts. However, many domains where it was thought that artificial intelligence could not compete with human experts turned out that the assumption was wrong. Especially when think about the investing, decision making is backed by huge amount of data, processing that much data is where computers excel at, not humans. One could easily imagine algorithms running on computer someday out perform in profits for investments than the likes of Warren Buffett or Peter Lynch, just like what AlphaGo did to Lee Sedol, the south Korean Go champion. Indeed, there are many current ongoing attempts on computerized investing. Two Sigma is one of those companies who is using machine learning to invest.

Recently, a Kaggle competition that was hosted by Two Sigma challenged machine learning practitioners to predict the movements of stock prices in ten days using market and news data. By participating in this competition, the author tried both deep learning and gradient boosting decision tree models and positioned rather in the upper bracket judging by public leader board of the competition. The implication of this result is machine learning can be effective even if it's done by a regular bachelor student in engineering field. It does not take PhD or teams of machine learning veterans to come up with nice results.
Thus, in this thesis the author aims to lay out the process and learnings of his Kaggle competition.

Keywords	Fintech, data science, machine learning, Kaggle, GBDT, big data

Contents

List of Abbreviations

API	Application programming interface (API). A protocol of communication methods among various components. An API could be a web API, operating system API, database system API, computer hardware API, or software library API.
GBDT	Gradient Boosted Decision Trees (GBDT) is a machine learning algorithm that iteratively constructs an ensemble of weaker decision trees by boosting.
NN	A neural network (NN) is a network of neurons, or in terms of AI, it means an artificial neural network, composed of artificial neurons which solve artificial intelligence (AI) problems.
XGB	XGBoost. Scalable, Portable and Distributed Gradient Boosting (GBDT, GBRT or GBM) Library, for including but not limited to R, Java, Python, Scala, C++. Runs on one or more machines in distributed manner e.g. in Hadoop, Spark

1 Introduction

Data is everywhere nowadays. Emerging data science and machine learning enabled us to make sense of huge data that used to be impossible or extremely difficult to interpret. One example of it is financial data and stock market. Everyday there will be countless financial, economic, business news from myriads of news company as well as data from stock exchanges about prices of instruments, trading volumes, returns over certain periods, etc.

Usage of machine learning techniques on large data is called ‘data mining’. The term comes from idea that earth and raw material are thought of as raw data, precious material after processing are thought of as a model for predictions with high accuracy. The data mining will revolutionize the ways human make decisions in many fields including finance industry. [6, 25]

Previous to the boom of machine learning and big data, investing and trading used to be largely the area of human expertise. Legendary investors such as Warren Buffett, Peter Lynch, John “Jack” Bogle and Charlie Munger all are done brilliant works of analyzing huge information that are relevant for their strategies and achieved great successes. While there is no saying when computerized investing will outrun the aforementioned investing gurus, several investment institutions of today including BlackRock, TwoSigma are powered by big data and machine learning already. Investing AI of today still maybe not a match for individual investing gurus yet, however, as one day Alpha Go surprised the world by defeating South Korean Go champion, the day might come in the future AI out performs very talented human investors. That is, however, not the purpose of this particular thesis. As machine learning is becoming more and more common, there are various attempts in applying the technology in financial, or investing more specifically, sector. The Kaggle - a web site for machine learning experts and data scientists that hosts various challenges and competitions - competition that is going to be explained in this thesis can also be understood in the same context.

Various institutions in the world tries to utilize big data and profit from doing it. In the similar light, there was a Kaggle competition hosted by ‘Two sigma’ regarding these very financial news and market data. It was about using market and news data to predict stock price movement in 10 days from an observation. The market data is provided by Intrinio which is a platform that provides market data API for developers. The news data is provided by Thomson Reuters which is one of the world’s most trusted provider of answers. The Author had an opportunity to participate in the competition and ranked top 10% on the leader board. There were plenty of learnings that were obtained from the experience and sharing them is what this thesis aims to.

2 Data provided by Host [1]

2.1 Market data

The market data provides a part of US instruments. The included instruments constantly change and is determined by the trading volume and availability. The data contains returns calculated over a variety of time periods. These properties are assumed for returns:

- They are calculated either from the opening time to the open of another trading day or from the closing time to the close of another.

- Returns can be raw, that is the data is not modified, or residual(Mktres), that is movement of the whole market is accounted for when calculating returns, resulting in only movements that are inherent of an instrument.

- Although returns can be over any interval. Only one day and ten-day intervals exist in the provided market data.

- Returns are prefixed with 'Prev' if they are related to past, or 'Next' if related to future.

Following features exist in the market data:

- ‘time’ - the current time in UTC (Coordinated Universal Time)

- ‘assetCode’ – an identifier of an asset

- ‘assetName’ - name for a multiple assetCodes. "Unknown" is used as value if assetCode column does not have any data in the news data.
- ‘universe’ - Whether or not the asset on that day is accounted for when scoring. Provided only in the training time period.
- ‘volume’ – the amount of that a certain asset code is traded on the trading day
- ‘close’ - the price at the time of closing of a trading day
- ‘open’- the price at the time of opening of a trading day
- ‘returnClosePrevRaw1’
- ‘returnOpenPrevRaw1’
- ‘returnClosePrevMktres1’
- ‘returnOpenPrevMktres1’
- ‘returnClosePrevRaw10’
- ‘returnOpenPrevRaw10’
- ‘returnClosePrevMktres10’
- ‘returnOpenPrevMktres10’
- ‘returnOpenNextMktres10’(float64) - ten-day, residual return. The dependent variable (y) for the competition. It is guaranteed that returnsOpenNextMktres10 has non-null value.

2.2 News data

The news data has data on both the news level and asset level.

- ‘time’ - timestamp of the time when the data was available on the feed (second precision).
- ‘sourceTime’ - time at the creation of the news
- ‘firstCreated’ - timestamp at the creation of the first version of the item when there are more than one versions.
- ‘sourceId’ - an Id of a news item.
- ‘headline’ - the news item's headline.
- ‘urgency’ - differentiates story types by degree of urgency (from 1 being the most urgent to 3 being regular news).
- ‘takeSequence’ - take sequence no. of a news item, counting from 1.
- ‘provider’ - identifier of the provider of news items (for example, RTRS for Reuters News).
- ‘subjects’ - topic codes and company codes related to the news item. Topic codes represent the kind of news item's subject. They represent asset classes, geo-graphical regions, events, industries, and more.
- ‘audiences’ - identifies the desktop news (s) the news belongs to. Usually tailored to certain group of news consumers.
- ‘bodySize’ - the size of the body in number of characters.
- ‘companyCount’ - the count of companies mentioned.

- ‘headlineTag’ - the headline of the news.
- ‘marketCommentary’ – flag if the news discusses general market, e.g. after trading hour summaries.
- ‘sentenceCount’ - the count of sentences in the news.
- ‘wordCount’ - the total count of words.
- ‘assetCodes’ - set of assets included in the item. Unlike from the market data which has ‘assetCode’ as a singular form for id of the row, it is plural.
- ‘assetName’ – asset name.
- ‘firstMentionSentence’ - the first sentence in which the asset is mentioned.
    - 1: headline
    - 2: mentioned in body, first sentence
    - 3: mentioned in body, second or later
    - 0: the asset was not mentioned.
- ‘relevance’ - a decimal number indicating the relevance of the news. It is a value between 0 to 1. The relevance is 1 when the asset is mentioned in the headline. When the item has the alert type as the ‘urgency’, relevance is estimated by ‘firstMentionSentence’.
- ‘sentimentClass’ - indicates the dominant class of sentiment for the news. The one has the highest probability is assigned as the class.
- ‘sentimentNegative’ - probability that the news is negative.
- ‘sentimentNeutral’ - probability that the news is neutral.

- ‘sentimentPositive’ - probability that the news is positive.
- ‘sentimentWordCount’ - the total count of words that are relevant to the asset. With ‘wordCount’, it can decide the proportion of the news relevant to asset.
- ‘noveltyCount12H’ - The twelve-hour novelty of content within news.
- ‘noveltyCount24H’ - likewise, but for twenty-four hours.
- ‘noveltyCount3D’ - likewise, but for three days.
- ‘noveltyCount5D’ - likewise, but for five days.
- ‘noveltyCount7D’ - likewise, but for seven days.
- ‘volumeCounts12H’ - the twelve -hour volume of news per asset.
- ‘volumeCounts24H’ - likewise, but for twenty-four hours.
- ‘volumeCounts3D’ - likewise, but for three days.
- ‘volumeCounts5D’ - likewise, but for five days.
- ‘volumeCounts7D’ - likewise, but for seven days.

2.3 Overview of training data set

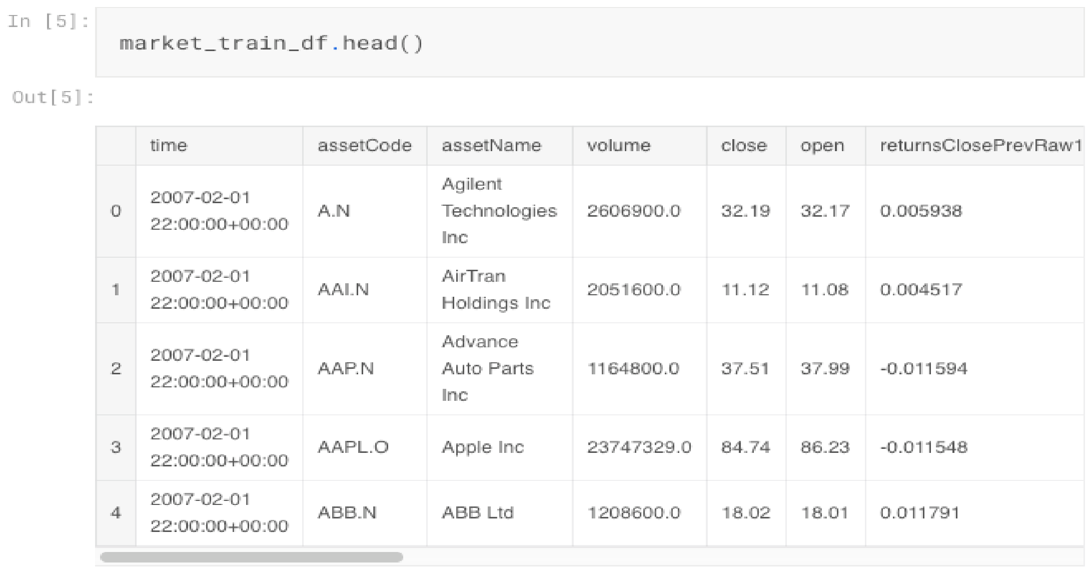
Host provides their own custom **kaggle.competitions.twosigmanews** Python module. It is used to control the flow of information to ensure that participants are not using future data to make predictions for the current trading day.

```
from kaggle.competitions import twosigmanews

env = twosigmanews.make_env()

(market_train_df, news_train_df) = env.get_training_data()
```

```
print(f'{market_train_df.shape[0]} samples and {market_train_df.shape[1]} fea-
tures in the training market dataset.')
=> 4072956 samples and 16 features in the training market dataset.
```

Listing 1. Import twosigmanews module, load the training data, output the number of rows and columns.

Examining the first 5 rows of shows that the earliest data in our training set begins from 01-02-2007.



```
In [5]:
market_train_df.head()

Out[5]:
```

	time	assetCode	assetName	volume	close	open	returnsClosePrevRaw1
0	2007-02-01 22:00:00+00:00	A.N	Agilent Technologies Inc	2606900.0	32.19	32.17	0.005938
1	2007-02-01 22:00:00+00:00	AAI.N	AirTran Holdings Inc	2051600.0	11.12	11.08	0.004517
2	2007-02-01 22:00:00+00:00	AAP.N	Advance Auto Parts Inc	1164800.0	37.51	37.99	-0.011594
3	2007-02-01 22:00:00+00:00	AAPL.O	Apple Inc	23747329.0	84.74	86.23	-0.011548
4	2007-02-01 22:00:00+00:00	ABB.N	ABB Ltd	1208600.0	18.02	18.01	0.011791

Figure 1. The first 5 rows from the market_train_df

On the other hand, checking by `tail()` method of pandas dataframe, the latest data is as of the end of 2016 which makes the dataset a decade worth of data. As shown in the Listing 1, there are about four million rows total. Pandas dataframe has a helpful function called describe() and info(). The former shows count, mean, standard deviation, min, 25th percentile, 50th percentile, 75th percentile and max values which can be useful in finding out the characteristic of dataset and finding out about outliers, the latter shows the data types of each column. Moving on to `news_train_data`, calling info method on it shows that there are more than nine million rows in the dataset and 35 columns. It clearly shows that there can be zero or more news about each asset on any given day.

3 Evaluation method used on competition

3.1 Score formula

In this competition, competitors must predict a signed confidence value between -1 to 1 times the market-adjusted return of ‘assetCode’ over a ten day window. If a machine learning model expects a stock to have a bigger positive return than the broad market over the next ten days, it might assign it a large, positive ‘confidenceValue’ (near 1.0). If it expects a stock to have a negative return, it might assign it a negative value. If unsure, you might assign it a value near zero.

During the evaluation period, the following is calculated:

$$x_t = \sum_i \hat{y}_{ti} r_{ti} u_{ti},$$

where r_{ti} is the ten-day return for day t for instrument i, and u_{ti} is a 0/1 universe variable that controls whether a particular asset is scored (market adjusted).

Submission score is the standard deviation over the mean of daily x_t values:

$$\text{score} = \frac{\bar{x}_t}{\sigma(x_t)}.$$

The score is 0 when the standard deviation of predictions is 0.

3.2 Submission File

You must make submissions directly from Kaggle Kernels. By adding your teammates as collaborators on a kernel, you can share and edit code privately with them.

The kernels environment automatically formats and creates your submission files in this competition when calling env.write_submission_file(). There is no need to manually create your submissions. Submissions will have the following format:

```
time,assetCode,confidenceValue
2017-01-03,RPXC.O,0.1
```

```
2017-01-04,RPXC.O,0.02
2017-01-05,RPXC.O,-0.3
etc.
```

Listing 2. Two Sigma competition submit format

4 Feature engineering

Feature engineering techniques depend on the models that are used to train and to inference.

4.1 Feature engineering for linear models or neural network (NN) models

Linear models and neural network (NN) models cannot handle non-numerical features as they are. They have to be transformed before being fed into models using techniques such as one hot encoding or embedding. These models work in a way that models make predictions by applying matrix multiplications and adding bias terms to input features. The predictions are then used to get the loss from the loss function. And derivatives for each parameter (weight or bias) is calculated and the parameters are updated by subtracting the product of learning rate α and the derivative with respect to the parameter that is being updated. If there are big differences in scales of parameters, the update will be mostly swayed by parameters of large range than by those with small range. Therefore, the all parameters are rescaled to have similar range. One common way to do this is make all parameters have mean of 0 and standard deviation of 1, and it is called standard score normalization. This is done by following formula.

$$\frac{X - \mu}{\sigma}$$

Another common way is to make all parameters in the range 0 – 1, and it is called feature scaling. This is done by following formula.

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

4.2 Feature engineering for decision-tree based models

On the other hand, if the model is decision-tree based models, categorical features have to be label encoded which means converting all categories to numerical representation. This can be done by, for example, giving unique numbers to all categorical values and map them. In the competition, ‘assetCode’ was label encoded in the following way.

```
def encode_assetCode(market_train):
    global code2array_idx
    market_train['assetCodeT']        =        market_train['as-
setCode'].map(code2array_idx)
    market_train = market_train.dropna(axis=0)
    return market_train

code2array_idx = dict(
    zip(market_train_df.assetCode.unique(),       np.arange(mar-
ket_train_df.assetCode.nunique()))
)

market_train_df = encode_assetCode(market_train_df)
```

Listing 3. Label encoding for assetCode column in market data

There are other ways to encode categorical values too. ‘Frequency encoding’ is one of them. It is done by mapping categories to their frequencies of belonging to a specific label.

Unlike non-tree-based models, tree-based models cannot find relationships between features that well. That is why it is important to generate features to help models learn better. Because the data we have is time series data, one usual technique that is used to generate features in time series data is generating lag/window features. Lag features are values at previous time steps. Window features are a summary of values over a fixed window of previous time steps [5].

Similar to real life, where technical investors make decisions based on values such as the peak/lowest price in 1 year, the difference between current price and its 1 month ‘Moving Average’, ‘Exponential Moving Average’, ‘Bollinger Band’, ‘The Relative Strength Index’, and ‘Volume Moving Average’, it makes sense to create window statistics of 1 week, 2 week, 1 month and 1 year or other timeframes. Common and simple descriptive statistics values like mean, median, max, min and exponentially weighted mean can be used to create features for each window. These window features are generated based on numeric columns in market data (because of 16gb RAM constraint, news data had to be used only in limited fashion).

Rows are identified with ‘assetCode’ and ‘time’. However, the numbers around specific assetCode do not mean much without the context of their usual range and their ratios to market mean (which also changes constantly). For example, if a stock’s opening price is 500 dollars today, can one tell if it is oversold or overbought? If the stock in question was Amazon, this would be insanely cheap because as of the beginning of 2019, it trades around 1600$. However, if it was a Tesla stock, it would be considered expensive than its usual price range because Tesla stock as of the beginning of 2019 trades around 300 ~ 350 $ range. Same principle applies to many other features such as raw returns (gain/loss which is not adjusted against any benchmark), (trading) volume. Therefore, it helps model to generate the ratio between these features and corresponding market means. Following is how you can generate market mean column for ‘volume’ and ‘close’.

```
BASE_FEATURES = ['returnsOpenPrevMktres10',
                 'returnsOpenPrevRaw10',
                 'open', 'close']
def add_market_mean_col(market_df):
    daily_market_mean_df = market_df.groupby('time').mean()
    daily_market_mean_df = daily_market_mean_df[['volume',
'close']]
    merged_df = market_df.merge(daily_market_mean_df,
                                left_on='time',
                                right_index=True,
                                suffixes=("",'_market_mean'))
    merged_df['volume/volume_market_mean'] = merged_df['volume']
/ merged_df['volume_market_mean']
    merged_df['close/close_market_mean'] = merged_df['close'] /
merged_df['close_market_mean']
    return merged_df.reset_index(drop = True)

BASE_FEATURES = BASE_FEATURES + ['volume',
                                 'volume/volume_market_mean',
                                 'close/close_market_mean']

market_train_df = add_market_mean_col(market_train_df)
market_train_df.head(3)
```

Listing 4. Generate market mean column for ‘volume’ and ‘closing’

In the same logic, price fluctuation on a single day could be an important factor of trading for day traders. Thus, the column ‘open/close’ which holds the ratio between ‘open’ / ‘close’ can be generated in the following way.

```
def generate_open_close_ratio(df):
    df['open/close'] = df['open'] / df['close']

BASE_FEATURES = BASE_FEATURES + ['open/close']

generate_open_close_ratio(market_train_df)
```

Listing 5. Generate market mean column for ‘volume’ and ‘closing’

Among the columns that is given in market data are raw return features. They are not that meaningful by themselves (unlike market residual features) because 1-dollar appreciation of an asset which has original price of 5 and that of 100 have very different implications. Raw return features’ ratio to 'open', 'close' would be more useful because it sees 1-dollar appreciation from 5 and from 100 very differently (20% and 1% each) and 1-dollar appreciation from 5 and 20 from 100 as the same (20% both).

```
open_raw_cols = ['returnsOpenPrevRaw1',
                  'returnsOpenPrevRaw10']
close_raw_cols = ['returnsClosePrevRaw1',
                   'returnsClosePrevRaw10']

def raw_features_to_ratio_features(df):
    for col in open_raw_cols:
        df[col + '/open' ] = df[col] / df['open']
    for col in close_raw_cols:
        df[col + '/close'] = df[col] / df['close']

BASE_FEATURES = BASE_FEATURES + ['returnsClosePrevRaw1/close',
'returnsClosePrevRaw10/close', 'returnsOpenPrevRaw1/open', 're-
turnsOpenPrevRaw10/open']

raw_features_to_ratio_features(market_train_df)
```

Listing 6. Generating the ratio columns between raw returns features and 'open', 'close'

All the columns that are now stored in 'BASE_FEATURES' variable now can be used to generate window statistics features (hence the name, which means window statistics features are based on). Mean, median, max, min of windows of 5, 10, 20 and 252 business days are generated with the code below.

```
global N_WINDOW, BASE_FEATURES
N_WINDOW = np.sort([5, 10, 20, 252])

global FILLNA
FILLNA = -99999

ewm = pd.Series.ewm

def generate_features_for_df_by_assetCode(df_by_code):
    prevlag = 1
    for window in N_WINDOW:
        rolled = df_by_code[BASE_FEATURES].shift(prevlag).roll-
ing(window=window)
```

```
        df_by_code = df_by_code.join(rolled.mean().add_suf-
fix(f'_window_{window}_mean'))
        df_by_code = df_by_code.join(rolled.median().add_suf-
fix(f'_window_{window}_median'))
        df_by_code = df_by_code.join(rolled.max().add_suf-
fix(f'_window_{window}_max'))
        df_by_code = df_by_code.join(rolled.min().add_suf-
fix(f'_window_{window}_min'))
        for col in BASE_FEATURES: # not sure if this can be op-
timized without using for loop but I only know how to calculate
exponentially moving averages like this
            df_by_code[col + f'_window_{window}_ewm'] =
ewm(df_by_code[col], span=window).mean().add_suffix(f'_win-
dow_{window}_ewm')
    return df_by_code.fillna(FILLNA)

def generate_features(df):
    global BASE_FEATURES, N_THREADS
    all_df = []
    df_codes = df.groupby('assetCode')
    df_codes = [df_code[1][['time','assetCode'] + BASE_FEATURES]
for df_code in df_codes]
    pool = Pool(N_THREADS)
    all_df = pool.map(generate_features_for_df_by_assetCode,
df_codes)
    new_df = pd.concat(all_df)
    new_df.drop(BASE_FEATURES,axis=1,inplace=True)
    pool.close()
    return new_df
new_df = generate_features(market_train_df)
market_train_df = pd.merge(market_train_df, new_df, how =
'left', on = ['time', 'assetCode'])
```

Listing 7. Generating lag/window feature columns based on ‘BASE_FEATURES’

The features generated for market data is now quite many. Due to memory constraint in the competition (16 GB) news data is not going to be used as much. The columns that are chosen to be kept are the hour part of ‘firstCreated’, count of ‘sentimentClass’ (which is meant to show how many news articles are written about specific asset code), mean of ‘sentimentClass’, ‘sentimentNegative’, ‘sentimentNeutral’ and ‘sentimentPositive’. News data can be merged to market data by ‘time’ and ‘assetName’ column. ‘assetCodes’ column is not suitable to be a column for join as it contains multiple asset codes. Even though ‘assetCode’ is one of the key columns in market data, it is still fine to use ‘assetName’ in key column instead because asset codes with same asset name are related assets (for example, normal stock and preferred stock from a same company) After dropping all the unnecessary columns from news data following code was used to generate news related features and merge them to the market data.

```
def merge_with_news_data(market_df, news_df):
    news_df['firstCreated'] = news_df.firstCreated.dt.hour
    news_df['assetCodesLen'] = news_df['assetCodes'].map(lambda
x: len(eval(x)))
    news_df['asset_sentiment_count'] = news_df.groupby(['asset-
Name', 'sentimentClass'])['firstCreated'].transform('count')
    kcol = ['time', 'assetName']
    news_df = news_df.groupby(kcol, as_index=False).mean()
    market_df = pd.merge(market_df, news_df, how='left',
on=kcol, suffixes=("", "_news"))
    return market_df

market_train_df = merge_with_news_data(market_train_df,
news_train_df)
del news_train_df
```

Listing 8.	Generating news features and merging them to existing market data

Next up features that are useless in learning such as id column, dependent variable, timestamps or hard to process columns (because they need natural language processing first) such as subjects are filtered out.

```
fcol = [c for c in market_train_df if c not in ['assetCode', 'assetCodes', 'as-
setCodesLen', 'assetName', 'audiences',
                                                'firstCreated', 'headline', 'headlineTag',
'marketCommentary', 'provider',
                                                'returnsOpenNextMktres10', 'sourceId',
'subjects', 'time', 'time_x', 'universe','sourceTimestamp']]
```

Listing 9. Filtering out useless features

To prevent one feature to have too much sway than the other features it helps to adjust all features to be in similar range.

```
X = market_train_df[fcol].values
mins = np.min(X, axis=0)
maxs = np.max(X, axis=0)
range = maxs - mins
X = 1 - ((maxs - X) / range)
```

Listing 10. Scaling of features to be in a similar range

To evaluate that model is learning to generalize well for the new observations without overfitting, validation set is used to measure the loss. The ratio of training data and validation data depend on cases and how much data are available. However thirty percent is used quite often given there are enough data (> 10000).

```
X_train, X_test, up_train, up_test = model_selec-
tion.train_test_split(X, up, test_size=0.30, random_state=99)
```

Listing 11. Splitting training set and validation set with scikit-learn’s `model_selection`

Now that feature engineering is done and training and test data sets are prepared, it is the time to discuss about machine learning models.

5 Machine learning models

5.1 Common machine learning models

These models differ by the types of training data used, the way training data and the test data is used to evaluate the outcome.

- Supervised learning: Labeled examples are used as training data to make predictions for new observations. Typical use cases are classification, regression, and ranking problems.

- Unsupervised learning: Only unlabeled training data are used to make predictions for new observations. Clustering and dimensionality reduction are representative examples of use cases.

- Semi-supervised learning: Both labeled and unlabeled data are used to make predictions for new observations. It is useful when unlabeled data is cheap but labeled ones are pricy to obtain.

- On-line learning: Is done with multiple ongoing iterations. Training and testing phases are mixed. On every iteration, model predicts for a new observation and then will obtain the actual label calculating a loss.

- Reinforcement learning: The training and testing phases are mixed. The model chooses actions within the environment and then either be rewarded or punished. The model learns to maximize his reward after many rounds of iterations. The model makes choices whether to explore unknowns for more experiences versus making use of what is already learned. [7, 20-21]

Two types of supervised learning models were tried and submitted to the competition by author. One was a deep neural network model and the other was a gradient boosting decision tree (GBDT) model. Deep learning definitely is a model that is getting the most spotlight from the academia and media. It has become such a buzz word that is being talked about frequently in media. It is especially proven useful for unstructured data including but not limited to images, sounds. There are many deep-learning-based computer-vision models developed including Resnet, VGG. However, when it comes to structured (tabular) data gradient boosting decision tree models are proven to be very effective. The two types of submissions made by author also turned out GBDT model scored better than DNN model based on the public leader board (0.69608, 0.61938 were the best scores of submissions of each type).

5.2 Deep neural network model

Multiple nodes connected with directed links make up neural networks (NN). A link from node i to node j spreads activation from i to j. A numeric weight is associated to each link. Both the power and sign of the connection are determined by the weight. There are two types of neural networks. The connections in feed-forward type is unidirectional, which results in a directed acyclic graph. All nodes accept input from nodes in prior layer and emit output to nodes in later layer without the loop. This type of network represents a function of the input without any states internally except weights. However recurrent network sends its outputs again as its own inputs. Moreover, the output of the network is affected by its initial state, which depends on previous inputs. This property means that recurrent networks maintains short-term memory. [6, 728-729]

Deep learning models can be seen as multi-layered neural networks. Normal NNs have input layer, one hidden layer, and output layer. The only difference in deep learning models are they have more than one hidden layer. They need different ways of preprocessing than tree-based models. Tree-based models only need to be able to split at some arbitrary points (for example, the value of feature a is greater than x). But all features in deep learning models must be able to be multiplied by and added to. It is because DNN is essentially just a neural network (NN) model with more than one hidden layer. Input features will go through multiple matrix multiplications and additions with biases. One way to achieve this is one-hot encoding but when the cardinality (the number of possible values in a categorical feature) is high one-hot encoding is not memory efficient because every possible category means one more column will be added. In natural language processing (NLP) field, words are converted to embedded vector for this reason. This preprocessing technique is called word2vec. ‘assetName’ can be encoded with this word2vec technique for example.

After applying preprocessing technique that is applicable to DNN model, the specific architecture of DNN should be decided. The author has used PyTorch opensource library to define the model. PyTorch development is driven by Facebook. Inheriting `torch.nn.Module`, `FeedForwardNN` class is defined to receive Pandas table as the input and the output is the probability that a particular asset with 'assetCode' will appreciate in value in the next 10 days. After the input layer, fully-connected linear layer, batchnorm2d layer and dropout layer were repeatedly used. The number of parameters in linear layer is reduced layer by layer. Dropout rate of 0.1 is set and Adam optimizer is used. Various learning rate was tried but setting it as 0.21 yielded the best result in terms of score. 0.21 learning rate is unusually high value but if combined with learning rate scheduler it can yield a good result. A learning rate scheduler reduces learning rate as training progresses by specified number of epochs. This scheduling works well for the stochastic gradient descent (SGD) algorithm because in the early stage of training parameters are far from optimal so they can be updated by bigger values. But this is no longer true when parameters are tuned. Large learning rate prevents parameters from being converged. That is why starting training with large learning rate and decreasing it as training progresses work well. Cross entropy loss was chosen to be the loss function so that models can obtain gradient.

```
class FeedForwardNN(nn.Module):
    def __init__(self, lin_layer_sizes, lin_layer_dropout_rate,
                 output_size=1, emb_dropout=0, emb_dims = {},
no_of_cont = 0):
        super(FeedForwardNN, self).__init__()

        # Embedding layers
        self.emb_layers = nn.ModuleList([nn.Embedding(vocab,
dim)
                                          for vocab, dim in
emb_dims])

        self.no_of_embs = sum([dim for vocab, dim in emb_dims])
        self.no_of_cont = no_of_cont

        # Linear Layers
        first_lin_layer = nn.Linear(self.no_of_embs +
self.no_of_cont,
                                    lin_layer_sizes[0])

        self.lin_layers =\
         nn.ModuleList([first_lin_layer] +\
              [nn.Linear(lin_layer_sizes[i], lin_layer_sizes[i +
1])
               for i in range(len(lin_layer_sizes) - 1)])

        for lin_layer in self.lin_layers:
            nn.init.kaiming_normal_(lin_layer.weight.data)

        # Output Layer
        self.output_layer = nn.Linear(lin_layer_sizes[-1],
                                      output_size)
        nn.init.kaiming_normal_(self.output_layer.weight.data)

        # Batch Norm Layers
        self.first_bn_layer = nn.BatchNorm1d(self.no_of_cont)
        self.bn_layers = nn.ModuleList([nn.BatchNorm1d(size)
                                        for size in
lin_layer_sizes])
        # Dropout Layers
        self.emb_dropout_layer = nn.Dropout(emb_dropout)
        ####
        self.droput_layers = nn.ModuleList([nn.Drop-
out(lin_layer_dropout_rate) for size in lin_layer_sizes])
        # self.droput_layers = nn.ModuleList([nn.Dropout(size)
for size in lin_layer_dropouts])
        ####

    def forward(self, cont_data, cat_data, test = False):
        if self.no_of_embs != 0:
            x = [emb_layer(cat_data[:, i])
                 for i, emb_layer in enumerate(self.emb_layers)]
            x = torch.cat(x, 1)
            if not test:
                x = self.emb_dropout_layer(x)

        normalized_cont_data = self.first_bn_layer(cont_data)

        if self.no_of_embs != 0:
            x = torch.cat([x, normalized_cont_data], 1)
        else:
            x = normalized_cont_data

        if not test:
            for lin_layer, dropout_layer, bn_layer in\
                zip(self.lin_layers, self.droput_layers,
self.bn_layers):
                x = F.relu(lin_layer(x))
                x = bn_layer(x)
                x = dropout_layer(x)
        else:
            for lin_layer, bn_layer in\
                zip(self.lin_layers, self.bn_layers):
                x = F.relu(lin_layer(x))
                x = bn_layer(x)
        output = self.output_layer(x)
        return output

model = FeedForwardNN(no_of_cont=no_of_cont,
                      lin_layer_sizes=[
                          1500,
                          1300,
                          1000, 800,
                          550, 350,
                          220, 160,
                          140, 100,
                          70, 50,
                          40, 30,
                              20, 15,
                              12, 10,
                              8,   4
                              ],
                          output_size=2, emb_dropout=0.10,
                          lin_layer_dropout_rate=0.10)
optimizer = torch.optim.Adam(model.parameters(), lr=0.21)
exp_lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                                   step_size=2,
                                                   gamma=0.4)
criterion = nn.CrossEntropyLoss()
```

Listing 12. Defining DNN architecture, optimizer, learning rate scheduler and loss function

The actual training loop has to have two parts. The first of which is where it calculates loss and updates parameters (weights and biases) and the second of which is it validates against validation set but does not update parameters. The loop compares validation loss and keeps parameters only when new best validation loss is found. The following 'train_model' function shows this point.

```
def train_model(model, criterion, optimizer,
                train_dataloader, valid_dataloader,
                best_val_loss, num_epochs,
                scheduler):
    model.train()
    since = time.time()
    best_loss = best_val_loss
    best_model_wts = copy.deepcopy(model.state_dict())
    for epoch in range(num_epochs):
        scheduler.step()
        # zero the parameter gradients
        train_running_loss = 0.0
        for labels, cont_x, cat_x in train_dataloader:
            current_batch_size = labels.size()[0]
            if current_batch_size == 1:
                continue
            optimizer.zero_grad()
            with torch.set_grad_enabled(True):
                outputs = model(cont_x, cat_x)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()
                # statistics
            train_running_loss += loss.item()
        print('Train loss: {:.4f}'.format(train_running_loss /
len(train_dataloader)))

        valid_running_loss = 0.0
        for labels, cont_x, cat_x in valid_dataloader:
            current_batch_size = labels.size()[0]
            if current_batch_size == 1:
                continue
            # zero the parameter gradients
            # forward
            with torch.set_grad_enabled(False):
                outputs = model(cont_x, cat_x, True)
                loss = criterion(outputs, labels)
            valid_running_loss += loss.item()
        valid_epoch_loss = valid_running_loss / len(valid_data-
loader)

        # deep copy the model
        if valid_epoch_loss < best_loss:
            print(f'New best model found!')
            print(f'New record loss: {valid_epoch_loss}, previ-
ous record loss: {best_loss}')
            best_loss = valid_epoch_loss
            best_model_wts = copy.deepcopy(model.state_dict())

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.for-
mat(time_elapsed // 60, time_elapsed % 60))
```

```
    print('Best loss: {:.4f}'.format(best_loss))
    model.load_state_dict(best_model_wts)
    return model, best_loss
```

Listing 13. ‘train_model’ function where only parameters with the least loss are kept

Tweaking the number of layers and other hyper parameters yielded unreliable results on the public leader board. The best score was 0.61938 which was achieved by using the code from Listing 9, however many times scores were somewhere between 0.5 and 0.6.

5.3 Gradient boosting decision tree model

Decision tree-based models are both one of the simplest and yet the most powerful ways of machine learning. A decision tree essentially is a function that receives input of one or more features and outputs a “decision”—a single value. The values are either discrete or continuous. A sequence of tests is performed for a decision tree to make its decisions. Every node in the decision tree represents a test of the value of one of the features. A specific value is returned by each leaf node in the tree. The decision tree algorithm feels natural considering many “How To” guides are essentially single decision trees stretched over longer instructions. [6, 697-698]

Gradient boosting is a way to do machine learning for both classification and regression problems, which builds a prediction model using an ensemble of weaker models, usually with decision trees. Although most of the Kaggle winners use either stack or ensemble of different models, one model that is part of most of the ensembles is some variation of Gradient Boosting (GBM) model. [2]

The concept of gradient boosting was proposed by Breiman that boosting can be thought of as an optimization logic on a proper loss function. Subsequently, Friedman developed the explicit regression gradient boosting. Boosting algorithms were viewed as iterative *functional gradient descent*. That is, algorithms that minimize the loss over function space by repeatedly choosing a function that is headed to the negative gradient. This view of boosting resulted in the emergence of boosting algorithms in many machine learning areas beyond just classification and regression.[3]

After preprocessing all the features to be numeric so that decision tree can be split at arbitrary values, the code to train them is easy. Hyperparameter tuning helps, so is ensemble of different models. ‘Hyperopt’ is one of many libraries for hyper parameter tuning. It is a Python library for serial and parallel optimization over search spaces. Prominent Python libraries based on gradient boosting algorithm include ‘XGBoost’, ‘CatBoost’ and ‘LightGBM’. Final submission used ‘LightGBM’ library because it trains faster than the others and also gave consistently good results. Training speed was important because the competition rule dictates submitted kernel can run only 9 hours maximum. The following code uses hyper parameters tuned in separate process and defines model with ‘LightGBM’ library.

```
hp_1 = [0.19000424246380565, 2452, 212, 328, 202]
hp_2 = [0.19016805202090095, 2583, 213, 312, 220]

params_1 = {
        'task': 'train',
        'boosting_type': 'gbdt',
        'objective': 'binary',
        'learning_rate': hp_1[0],
        'num_leaves': hp_1[1],
        'min_data_in_leaf': hp_1[2],
        'num_iteration': 250,
        'max_bin': hp_1[4],
        'verbose': 1
    }

params_2 = {
        'task': 'train',
        'boosting_type': 'gbdt',
        'objective': 'binary',
        'learning_rate': hp_2[0],
        'num_leaves': hp_2[1],
        'min_data_in_leaf': hp_2[2],
        'num_iteration': 350,
        'max_bin': hp_2[4],
        'verbose': 1
    }

gbm_1 = lgb.train(params_1,
        train_data,
        num_boost_round=500,
        valid_sets=test_data,
        verbose_eval=30,
        early_stopping_rounds=7,
        )

gbm_2 = lgb.train(params_2,
        train_data,
        num_boost_round=500,
        valid_sets=test_data,
        verbose_eval=30,
        early_stopping_rounds=7,
        )
```

Listing 14. Defining LightGBM model with tuned hyper parameters

Running the above code will output logs of training progress so that it is possible to verify if training is effective, if it is not overfitting, and what is the loss etc. The log outputs about ‘early_stopping_rounds’ which is the number of iterations that do not improve validation set score needed to stop training before specified ‘num_boost_round’ number of iterations is done. Then every ‘verbose_eval’ number of rounds, log output shows validation loss, which is ‘binary’ loss in this specific case. When training is finished, it tells if training was stopped early due to ‘early_stopping_rounds’ setting and also the best iteration, which is the iteration with lowest loss.

```
Training until validation scores don't improve for 7 rounds.
[10]      valid_0's binary_logloss: 0.624419
                         … (omit loggings)
Did not meet early stopping. Best iteration is:
[250]     valid_0's binary_logloss: 0.412581
```

Listing 15. Log output from the training process

The next step is to use the model that was just trained and predicting (inferencing) confidence values.

6 Inference phase

Trained model is used for predicting. This is called inferencing in another word. Inferencing is usually used in production to serve predictions or in the context of Kaggle competitions they are used to create solutions to submit. TwoSigma provides competitors with a module to get test data. 'env' is the variable which holds the module and the API to get test data is 'get_prediction_days()' method. The method returns iterator which returns one day's worth of data. Using the data competitors are asked to create Pandas dataframe with 'assetCode' and 'confidence' columns. The former is the identifier column, the latter is the prediction value. Once the prediction dataframe for a day is created, 'env' module's 'predict' method should be called with the prediction dataframe. Next day's data is obtainable only after 'predict' method is called. Lastly, 'env''s 'write_submission_file' method is used to write the submission file to the file system. The submission file contains predictions for all the days that was predicted with 'env''s 'predict' method. Note that test data need to go through the same data processing process as with the training data.

```
days = env.get_prediction_days()
total_market_obs_df = []
for (market_obs_df, news_obs_df, predictions_template_df) in
days:
    t = time.time()
    market_obs_df['time'] = market_obs_df['time'].dt.date
    return_features = ['returnsClosePrevMktres10','re-
turnsClosePrevRaw10','open','close']
    total_market_obs_df.append(market_obs_df)
    if len(total_market_obs_df)==1:
        history_df = total_market_obs_df[0]
    else:
        history_df = pd.concat(total_market_obs_df[-
(np.max(n_lag)+1):])
    new_df = generate_lag_features(history_df,n_lag=n_lag)
    market_obs_df = pd.merge(mar-
ket_obs_df,new_df,how='left',on=['time','assetCode'])
    market_obs_df = mis_impute(market_obs_df)
    market_obs_df = data_prep(market_obs_df)
    X_live = market_obs_df[fcol].values
    X_live = 1 - ((maxs - X_live) / rng)
    lp = (gbm_1.predict(X_live) + gbm_2.predict(X_live))/2
    confidence = lp
    confidence = (confidence-confidence.min())/(confi-
dence.max()-confidence.min())
    confidence = confidence * 2 - 1
    preds = pd.DataFrame({'assetCode':market_obs_df['as-
setCode'],'confidence':confidence})
    predictions_template_df = predictions_tem-
plate_df.merge(preds,how='left').drop('confidenceVal-
ue',axis=1).fillna(0).rename(columns={'confidence':'confi-
denceValue'})
    env.predict(predictions_template_df)

env.write_submission_file()
sub = pd.read_csv("submission.csv")
```

Listing 16. Code for inference phase

After running the code in Listing 12, submission file can be submitted from Kaggle notebook ‘Output’ section simply by selecting ‘submission.csv’ and clicking ‘Submit to Competition’. Kaggle gives feedback by means of the public score in about dozen seconds.

7 Final submission

Kaggle allows competitors to choose two kernels for the final submission. Among the two, only the one that results the best with private data set counts. Private leaderboard score is calculated the private data set that was hold out separately from calculating public data set. Public dataset and test dataset may be or may not be similar in characteristics (standard deviation, mean, etc). Because of the possible differences in public and private datasets, it is not always the best to choose submissions with the two best public scores, rather recommended way is to choose two quite different models (algorithm, hyper parameters, etc) with quite good score. Author's selection was LightGBM model where it has only one model, another selection was ensemble of two models. In terms of public score, the former has a slightly better score but that could be reversed in private score and private score is all that count. By the nature of this competition, private score will be calculated 5 months later because private test data is the current market and news data until 15th of July 2019 (As of writing this thesis, the time is February 2019).

8 Conclusion

Among all topics of data science, competitive data analysis is especially interesting. For an expert this is a great area to try his skills against other people and learn some new tricks; and for a beginner this is a good start to quickly and playfully learn basics of practical data science. For both, engaging in a competition is a good chance to expand the knowledge and get acquainted with new people.

To evaluate the result of using the techniques discussed in this thesis, leader board ranking is worth mentioning. Although, private score is not available as of the writing of thesis, using the method described in this thesis, author placed 264th place among 2897 teams. Many teams were made of more than one person, but author did not have any other team member. Considering the least number of team member, under top 10 percent is quite presentable result for today's Kaggle standard. Considering model architect was pretty simple (mostly because of the competition's computation resource restriction), most of the differences in score with other competitors can be attributed to the preprocessing method of data. Thus, lag/window data creation for time series data is a good way to create features and GBDT works well with tabular data sets.

The first limitation of this thesis is mainly that provided data has not been used as much as it could due to the limitations on computational resource that the competition imposed.

The second limitation is that due to the lack of time there are machine learning techniques that could be useful for the given problem but not tried. The reinforcement learning method and support vector machine are some of the models that could be tried and evaluated their usefulness if one decides to dig deeper in this topic.

References

1 D Sterling. Two Sigma: Using News to Predict Stock Movements[online]. Kaggle; 25 September 2018. URL: https://www.kaggle.com/c/two-sigma-financial-news/data. Accessed 11 February 2019.

2 Grover P. Gradient Boosting from scratch. Medium; 9 December 2017. URL: https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d. Accessed 4 February 2019.

3 LeDell E. useR! Machine Learning Tutorial. koalaverse.github.io; 5 April 2018. URL: https://koalaverse.github.io/machine-learning-in-R/index.html. Accessed 9 February 2019.

4 Brownlee J. Basic Feature Engineering with Time Series Data in Python. Machine Learning Mastery; 14 December 2016. URL: https://machinelearningmastery.com/basic-feature-engineering-time-series-data-python/?cv=1. Accessed 30 January 2019.

5 Russell J. S. and Norvig P. Artificial Intelligence: A Modern Approach, Third Edition. Essex: Pearson; 2009

6 Alpaydin E. Introduction to Machine Learning: Third Edition. Cambridge: The MIT press; 2014

7 Mohri M, Rostamizadeh A and Talwalkar A. Foundations of Machine Learning. Cambridge: The MIT press; 2012