Stat 6910 – 002

Deep Learning: Theory and Applications

Spring 2019

Final Project: High-Frequency Trade Sign Classification

Matt Isaac

Kegan Penovich

Jared Hansen

Due: Wednesday, May 1, 2019 at 5:00 pm

Outline

We will first describe the problem in terms of context and motivation, as well as the data we used in the training and testing process. We will then discuss the baseline models that were implemented. Next, neural network architectures and results will be described. Lastly, we will share our conclusions and indicate what future work might be done in this area.

Problem & Motivation

The objective of this project was to develop models for classifying high-frequency trade signs as either buyer-initiated or seller-initiated. There are several methods in the finance literature that have been developed for this purpose: (1) Lee and Ready's midpoint test, (2) Finucane's tick test, (3) Ellis, Michaely, and O'Hara's bid/ask test, and (4) Rosenthal's logistic regression model. A sampling of these accuracies from Rosenthal's 2012 paper Modeling Trade Direction is given in Table 1 below.

		Percent of trades correctly classified				
Sector	*N*	Modeled	EMO	LR.new	LR.old	Tick
Capital goods	216,800	**74.7**	73.0	72.1	71.8	61.6
Conglomerates	33,863	**84.7**	83.4	79.5	78.9	63.7
Cons. cyclical	236,193	**73.4**	72.1	71.7	71.4	62.9
Energy	228,978	**77.3**	76.1	73.3	72.9	62.5
Financial	1,014,479	**74.2**	72.4	72.4	72.2	63.3

Table 1: reported accuracies from Rosenthal's 2012 paper "Modeling Trade Direction".
Modeled = Rosenthal's logistic regression, EMO = bid/ask test, LR.new = midpoint test (new), LR.old = midpoint test (old), and Tick = the tick test.

Our goal is to surpass these accuracies by employing machine learning methods, particularly neural networks. Except for Rosenthal's model all these methods are simple decision rules that look at the quote (best bid and best ask) for a given trade to predict that trade's sign. By leveraging more sophisticated machine learning techniques we believe we can achieve much higher accuracies.

The motivation behind why we want to be able to predict trade direction accurately is to enhance information accessibility for researchers. Although we have access to ITCH data – an expensive data feed product provided by the Nasdaq exchange which contains trade signs – many do not. Instead many researchers and industry analysts have access to the NYSE's TAQ (trades and quotes) data which does not contain trade signs. Knowing the sign of each trade is very valuable information, allowing for accurate liquidity measures and price-estimation models. As such our goal is to provide a modeling approach that allows those with only TAQ access to predict trade signs with a high degree of accuracy. The first part of this process is to

develop models on labeled ITCH data; we can then employ transfer learning to predict onto TAQ data. For this project we focused on the first half of this process: developing models on the ITCH data.

Data: Pre-Processing & Description

We are working with ITCH data for 10 stocks across all 21 trading days in March 2018. For reference, the tickers of the stocks are: AAOI, BABY, CA, DAIO, EA, FANG, GABC, HA, IAC, and JACK. These stocks come from a variety of sectors including technology, healthcare, finance, and energy.

The ITCH data originally comes in text files which contain far more information than just the executed trades. A Python script was used to automate the data cleaning process, removing rows that weren't trades and columns that didn't lend predictive information (e.g. all values in a column were identical or contained only missing values). After the cleaning, these were the remaining features left in the data:

Feature	**Size**	**Type**	**Description**
Nanoseconds	10^13	int	Timestamp: nanoseconds since midnight of that trading day
Date	10^7	int	Given as "YYYYMMDD"
Match Num Order Num	10^5 10^7	Int int	Unique identifier, proxy for timestamp
Tracking Num	10^1	int	Nasdaq-specific, generally values of 2, 4, 6, or 8
Price	10^2	float (.4f)	The limit order price (can differ from execution price)
Exec_Price	10^2	float (.4f)	The price at which execution occurred
Shares	10^3	int	The limit order number of shares (can differ from ex_shares)
Exec_Shares	10^3	int	The number of shares executed for the trade
Message Type P	10^0	binary	Indicates that trade executed against a hidden order
Message Type E	10^0	binary	Generic 'order executed' message

An additional step taken in the data cleaning process was to standardize within each variable (except for the binary msg_type features) for each data set. The rationale behind this was that features larger in magnitude might have undue influence in prediction simply due to their size. As an illustrative example, to standardize the timestamp feature for a given data set we would do the following:

$$standardized(timestamp_i) = \frac{timestamp_i - mean_{timestamp}}{\sigma_{timestamp}} \ \forall i, i \in \{1, 2, ...len(dataframe)\}$$

Data: Aggregation

There were two ways that we considered aggregating the data for model development. The first was to treat each day within a ticker as a dataset and train/test the classifier algorithm on each day of data, thereby obtaining 21 test accuracies for each ticker. This will hereafter be referred to as the 'one ticker/one day' aggregation method. The other aggregation method we considered was to aggregate the data from all days within a single ticker, thus obtaining 1 test accuracy for each ticker. This aggregation will hereafter be referred to as the 'one ticker/all days' approach. These two aggregation methods are illustrated in Figure 1.

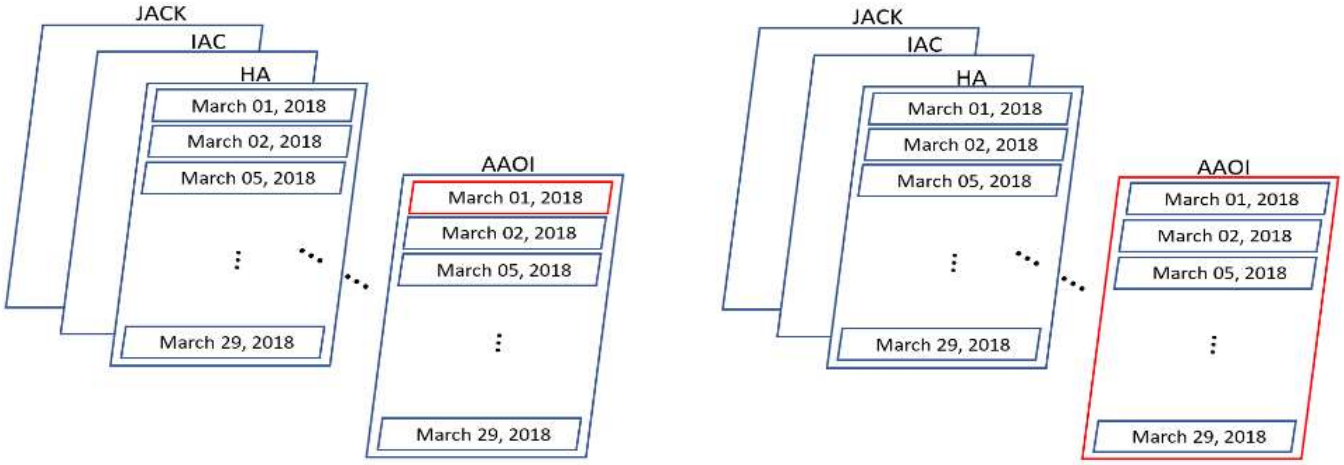

Figure 1: Aggregating data by 'one ticker/one day' (left) and by 'one ticker/all days' (right).

Baseline Models

Random forests and k-nearest neighbors (k-NN) were used to obtain a baseline accuracy that we could aim for when training the neural networks. These methods were used to classify the data aggregated by both the 'one ticker/one day' method and the 'one ticker/all days' method. Figure 2 shows the accuracy results from this process. The boxplots in Figure 2 show the accuracies from the 'one ticker/one day' aggregation, each boxplot being made up of the 21 training accuracies obtained from training on each of the 21 trading days. The solid line in Figure 2 shows the accuracies obtained by training random forests and k-NN on the 'one ticker/all days' data. It is clear that the random forests performed better overall than the k-NN algorithm.

We can also see that the test accuracies for some individual days are extremely high while others are quite low. Aggregating the data into the 'one ticker/all days' format gave us a fairly good 'middle of the road' accuracy compared to predicting individual days for a given ticker. Lastly, it appears that some tickers have signals that are more difficult to learn and classify than others. This is evident from the similar trends visible in the random forest accuracy line and the k-NN accuracy line. For instance, the dips in the predictive accuracy of the CA and HA tickers, as well as the spikes in predictive accuracy for the DAIO and IAC tickers occur for both the random forest and k-NN classifiers.

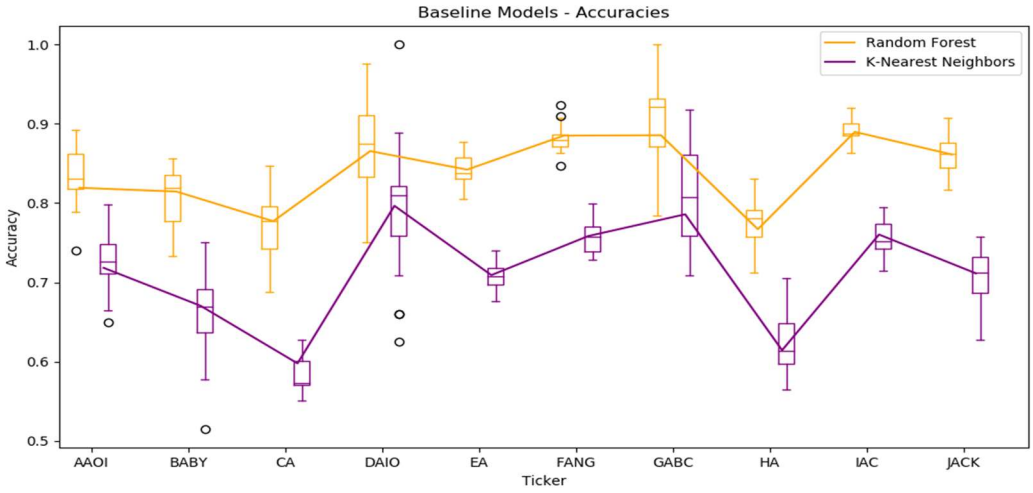

Figure 2: Test accuracies for baseline models. Boxplots for 'one ticker/one day' accuracies and solid line for 'one ticker/all days' accuracies.

NEURAL NETWORK MODELS: BIDIRECTIONAL LSTMs (B-LSTMs)

Architecture Selection

For the random forests and k-NN models, we treated the data as we would typical cross-sectional data. For the neural network models, we believe that we can get better prediction by more effectively utilizing the timestamp feature. Intuitively, for a given trade-of-interest (TOI), the immediately preceding and following trades contain a lot of valuable information. For example, if the 10 trades preceding TOI all have prices lower than the TOI and the 10 trades after it all have higher prices than the TOI, it is very likely that the TOI is a buy. Therefore, the inputs for a given TOI will be a sequence (in the example given, a sequence of 21 vectors where each vector is the feature information for a trade in the sequence).

The neural network architecture that is tailored toward sequence inputs is a recurrent neural network (RNN). Since the LSTM variant of RNNs have seen the most success and become the standard, this is the architecture we elected to use. Also, the LSTM architecture dovetails nicely with our notion of surrounding trades containing predictive information for the TOI. Since LSTMs “remember” both short-term and long-term dependencies they are an ideal fit for using immediately surrounding trades (short-term dependencies) and “remembering” patterns from earlier trades in the data (e.g. seeing a trade-run pattern early on in a sequence and applying similar prediction to a similar trade-run pattern later on in the sequence).

Since our task is binary classification, we use a single output neuron with a Sigmoid activation function. We round this output, giving a 0 or 1, which is then compared to the true signal (0 for buy, 1 for sell) and calculate the cost using the binary cross-entropy function.

Pre-processing Data for LSTM Input

A non-trivial portion of implementing an LSTM is pre-processing the data before feeding it into the network. Inputs are no longer just a 1-dimensional column vector as they might be for a feed-forward or convolutional network but are rather a sequence of vectors. Therefore we must do some work to partition and shape the data into correctly-sized inputs. Since our problem doesn't have the added complexity that a natural language processing problem would, namely that of having variable-length inputs (e.g. words of varying lengths) we don't have to do any kind of embedding to obtain vectors of uniform length. Figure 3 below gives a visual representation of how we might do this pre-processing.

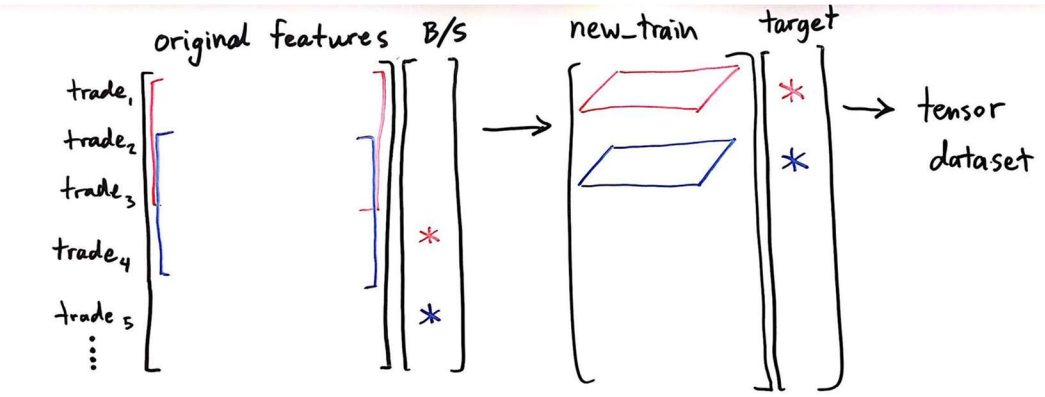

Figure 3: illustrating data pre-processing to allow for input to an LSTM architecture.

First, select a "window_size" to use (number of trades to consider) for generating each training sequence. In this example it is three; note the vertical length – in trades – of the red and blue brackets. In practice we used varying values for this, most commonly 10. Next select a "shift_by" value, which determines how many trades down we move to generate each new training sequence. In practice we elected to make this value one in order to preserve as much of the time dependence as possible. Now associate a response value with each training sequence. In Figure 3 these are denoted by asterisks, with the red training input corresponding to the red asterisk (buy/sell response). The 2-dimensional training input array now becomes a single element in the new_train (new training) data set, and the corresponding binary response is the associated target value. Then we bind this new_train array and target array into a tensor data set. At this point the data is in a format such that we can feed it into an LSTM for training and testing.

Keras Implementation

Initially no data standardization was done before model training and a unidirectional LSTM was used. This yielded test accuracies in the range of about 52-56%. No amount of parameter tuning or increasing the number of training epochs changed this. After standardizing the data, test accuracies moved up to the mid-60s. Next, since it was much easier to do than we'd anticipated, we implemented a B-LSTM. The only additional coding required was changing the value of a single argument, which was a very pleasant surprise.

However, this change alone only increased accuracy by about 1%. Parameter tuning didn't really improve accuracy, so we opted to increase the number of training epochs to 75. As can be seen in Figure 4a (left), this did help to increase accuracy but led to some dramatic overfitting. To combat this we introduced L2 regularization and dropout which significantly helped mitigate overfitting, illustrated in Figure 4b (right).

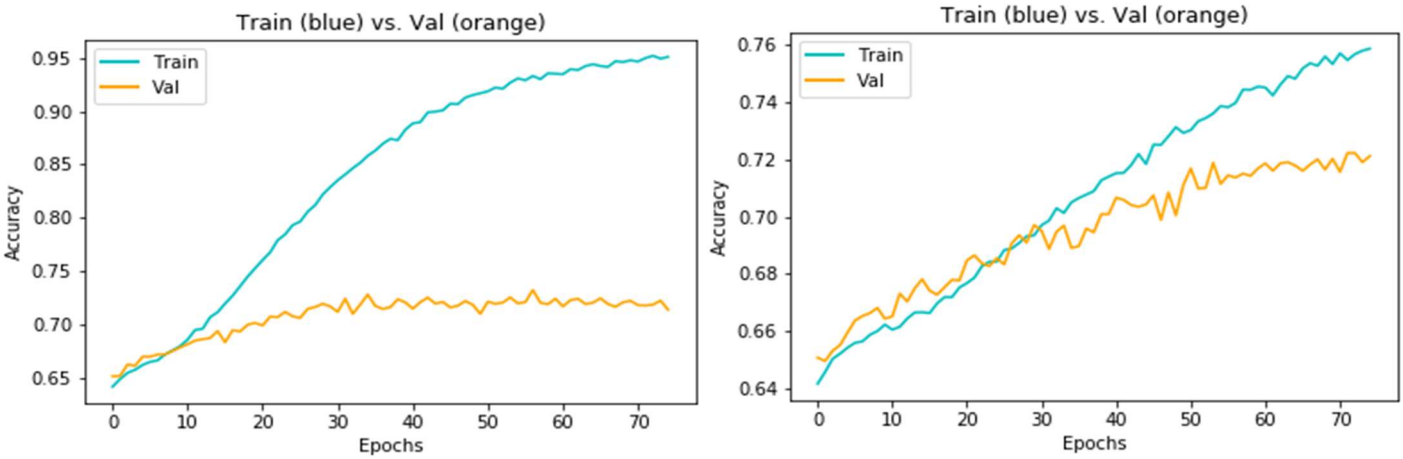

Figure 4a, 4b: Keras model without (left) and with (right) regularization and dropout

Technical specifications of the final Keras model included: a single LSTM layer with 100 neurons, the Adam optimizer (default parameters), learning rate = 0.01, dropout = 0.2, batch size = 64, L2 parameter = 0.9, and number of training epochs = 75.

PyTorch Implementation

We also worked with PyTorch (PT) to develop a B-LSTM model. Initially we started with a vanilla feed-forward network in PT, achieving about 60% test accuracy. Upon realizing that an RNN would be a better-suited architecture that accuracy rate bumped up to about 63-64% using a unidirectional LSTM. After implementing a B-LSTM on standardized data, accuracy increased to a level just below that of the Keras model, getting consistently between 64-70% accuracy. This can be seen in Figure 5 below.

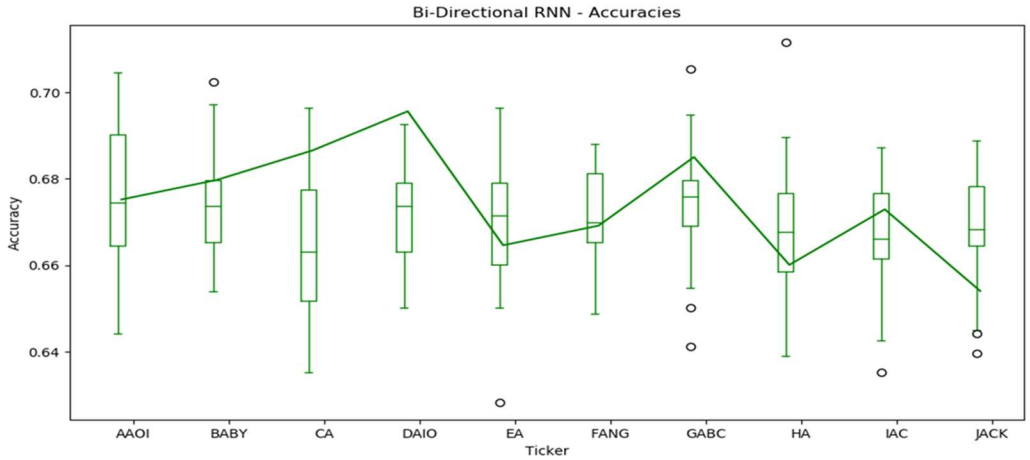

Figure 5: test accuracies across tickers for PyTorch B-LSTM model. Boxplots show test accuracies for the 'one ticker/one day' aggregation method while the solid line shows test accuracy for 'one ticker/all days' aggregation method.

Technical specifications of the final PT model included: two LSTM layer each having 128 neurons, the Adam optimizer (default parameters), learning rate = 0.003, batch size = 20, and number of epochs = 7. Since we didn't see a large increase in training accuracy in the Keras model from increasing the number of training epochs, we left this as a lower value and didn't worry about regularizing. This helped from both a computational time perspective, as well as a "one less parameter to tune" perspective since we had to fit many models to generate the Figure 5 graphic.

Conclusions/Discussion

Overall, the random forest classifiers outperformed the neural networks that were implemented. Figure 7 gives a final graphical comparison of forests versus the network models made in Keras and PyTorch.

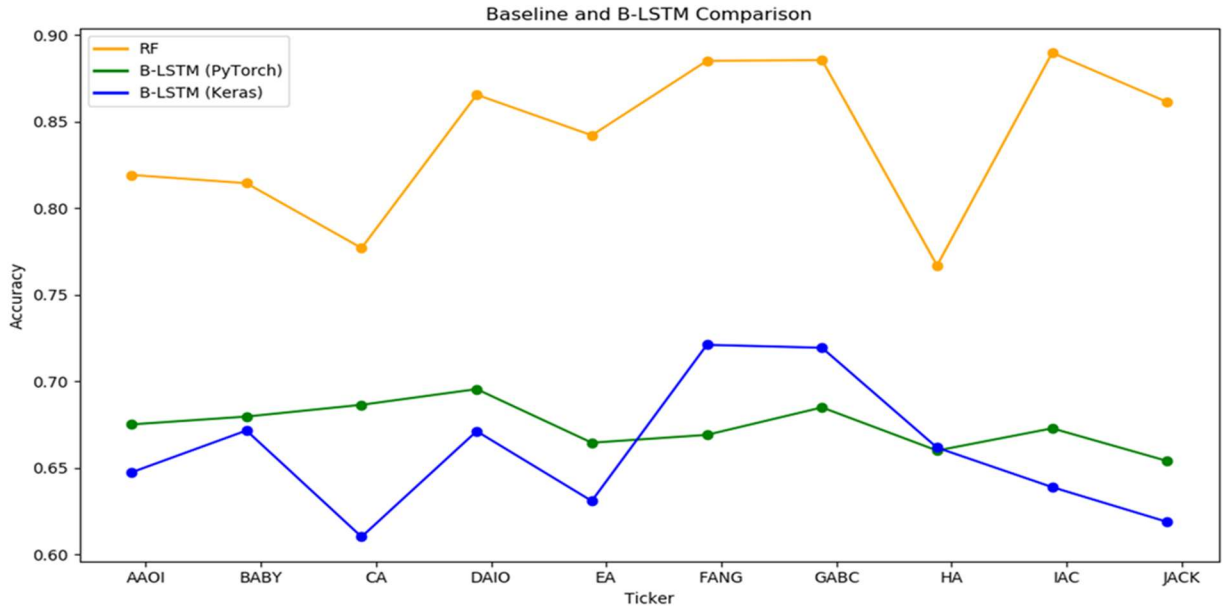

Figure 6: compared accuracies for forests and the B-LSTM models. We can see that forests performed best, and the two network models gave similar accuracies to each other. We can also see that some tickers are easier to characterize than others, regardless of the model used.

Random forests required a lower investment of effort and time than the B-LSTMs, for a much higher return in terms of predictive accuracy. However, we do not believe that B-LSTMs cannot be successfully applied to this problem; rather, we think that more time needs to be dedicated to exploring possible architectures, learning rates, and other parameters. It seems that some form of RNN would be well-suited to this problem due to the importance of the time aspect in classification. In the variable importance plot generated from the random forest algorithm, shown in Figure 7, the three most important variables are order_number, match_number, and nanoseconds. Nanoseconds is the timestamp associated with these data, and order_number and match_number are both a sort of proxy for the timestamp (they are monotonically increasing integers, with this increase corresponding to an increase in nanoseconds). Thus, we suspect that the best results will likely come from a neural network which best characterizes the time aspect of these data.

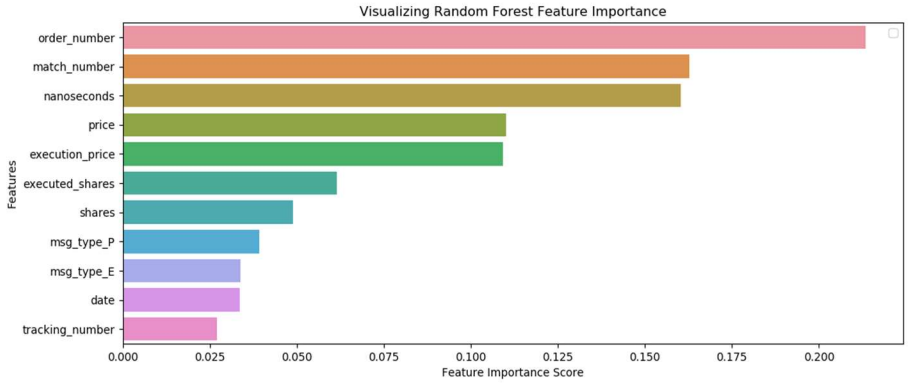

Figure 7: Random Forest Variable Importance Plot. Note that the top three variables are either the timestamp (nanoseconds) or proxies for the timestamp (order_number and match_number).

Future Work

As was mentioned in the *Problem & Motivation* section, the ultimate goal of this project is to train models that can accurately predict onto TAQ data. To this end, once neural network models that achieve higher accuracy on ITCH data are developed, the task of transfer learning onto the TAQ data will be undertaken. This will involve matching up trades across corresponding ITCH and TAQ data sets, and then using pre-trained models to see how well we can do on just the TAQ data.

Also, there is the possibility of some additional, interesting application of neural networks. The quotes portion of the TAQ data has far more observations than there are for the corresponding trades data. This is because not every quote (best bid and best ask) results in a trade. The canonical approach has been to try and match each trade with a single quote, attempting to account for asynchronous matching of quotes and trades (trades are reported with a delay relative to the corresponding quote). This, in and of itself, is a modeling problem of import. However, an approach that might be used to turn this problem into an opportunity would be to use a hierarchical B-LSTM. One LSTM would take in quote data (multiple quotes as an input sequence), then pass output to another LSTM. This second LSTM would take hidden and cell states from the first LSTM, as well as input sequences in the form of trades, and would be used to predict the sign for a given trade. Intuitively, the quote data that doesn't result in a trade still contains predictive information. For example, if the bid-ask spread is narrowing over time and is narrowing toward the best bid or the best ask, this is giving an indication as to the sign of the next trade. An LSTM architecture would be a way of capturing these patterns.

Contributions

Matt

- Fit baseline models and obtained baseline accuracies
- Developed code to standardize data within features across each dataset
- Lead out in progress report compilation/submission
- Experimented with feed-forward and B-LSTM networks
- Create visualizations of baseline and neural net accuracies for presentation and report
- Wrote sections relating to contributions for final report

Kegan

- Did research on best architectures for time-dependent data and how to implement them
- Developed code using Keras to pre-process data into tensors and implement LSTM and B-LSTM models
- Extensive tuning of B-LSTM models, including experimentation with hyperparameters and architecture
- Created visualizations of Keras LSTM models overfitting and responding to regularization/dropout
- Wrote sections relating to contributions for final report

Jared

- Lead in developing project proposal and coordinating assignments for the project
- Developed code to automate the data cleaning
- Created an initial random forests model, did some basic EDA
- Developed code using PyTorch to pre-process data into tensors and implement LSTM and B-LSTM models
- Oversaw development of the project presentation, as well as compiling and writing of the final report
- Wrote sections relating to contributions for final report