Financial Data Acquisition for Predictive Analysis

Final Project Report
W205 Storing and Retrieving Data - Spring 2017

By Shane Conner, Ben Attix and Jeffrey Hsu

Research Problem

A single business day hosts millions of trades across a multitude of markets and in terms of data structuring, each exchange is transactional data. Investors also have data for an asset's quarterly and annual income statements, cash flow, balance sheets, etc. Additionally, financial institutions, news, and research firms offer qualitative data and analysis which are often presented in grades or ranks. Simply put, the amount of collective information makes the stock market an incredibly complex industry.

As a result, financial firms utilize quantitative analysis to help distinguish the signal from the noise. Accordingly, a predictive model that is accurate, consistent, and unbiased is a desirable tool for any investor. However, to build a predictive model there are two primary components required - historic data to base a model off of and real-time data to apply the predictive model to.

There are several resources which supply rich stock market data, however this informations come at a steep premium - especially for a single user. For example, the top supplier of financial information, Bloomberg, charges a single user $24,000 per year to access data on their *Bloomberg Terminal* platform. Their closest competitor, Thomson Reuters Eikon, charges $21,600 per year with a bare bones version at $3,600 per year. While this cost is negligible to a financial institution, it is unfeasible for the majority of individual investors.

The objective of our project is to obtain both historical and real-time stock market data by aggregating financial data from a variety of sources. With the historical data, we will be able to perform exploratory data analysis that provides insights to variable relationships and ways to enrich the data. Additionally, the historical data we collect will be used as training data for a model that can be applied to real-time financial data and predict an asset's return on investment.

Data Architecture

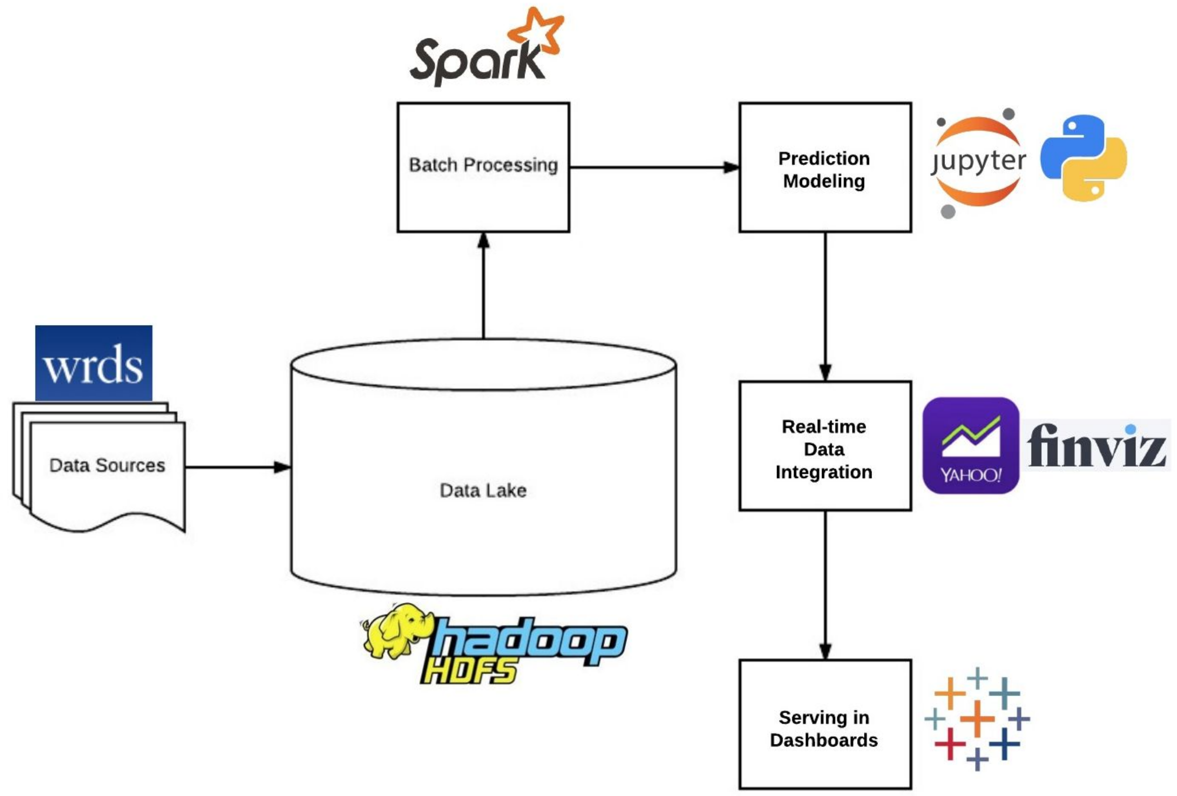
We have selected our data architecture mostly based on the characteristics of the datasets incorporated and with our anticipated processing needs. The datasets we selected come with large amount of stock features. A subset of stock features is over 200 variables. This adds complexity on the processing phase of applying schema, transformation and cleaning data. Also, in order to provide a holistic view on the stock performance, we incorporate multiple different datasets from different sources in order to avoid bias from single source data. To address these characteristics from the source datasets, the data architecture needs to handle stock data in various formats, sources and types. Flexibility on incorporating new data sources is needed since we wouldn't know beforehand the semantics of the data, schemas to impose and stock features useful in the analytical phase. If new data features is found to be needed in the machine learning stage, we will need to incorporate new dataset without changing the architecture, the existing data pipelines or processing implementations. In terms of the processing needs, raw datasets were read with schema, merged, transformed and cleaned to create the historical dataset required for training the initial machine learning model. The historical datasets are formed by merging financial ratios, CRSP security monthly, Beta suite, stock link table and recommendations datasets. The unique key is formed by combining GVKEY and the month when the stock price was reported. The unique key of “GVKEY-year-month” is used for merging raw datasets. Several procedures are carried out to transform the merged dataset. For instance, grouping by firm (GVKEY) and sort by the month of the record or creating Z-score of the numeric stock features which are also included in the machine learning models. Cleaning is performed by filling the missing columns with records of the next or previous row given the datasets are already sorted. All these procedures on transformation and filling missing data requires heavy processing power, especially performing on a large dataset we formulated. Thus the processing pipeline requires the possibility to scale out in the face of growing data size.

Based on the needs, we’ve selected an ELT data lake architecture. The primary data lake is based on HDFS for storing raw data and transformed data. The main benefit of a data lake architecture is that it allows us to acquire the raw datasets into our data store without the need to apply schema on it. Each of our datasets comes with different structures. Before performing exploratory data analysis, we wouldn’t know which schema and transformation we want to apply to the data. Thus, we want to ingest the raw data into the data lake, and apply schema on read when we want to batch process them. On each stage of the data processing, output result is materialized on the data lake for persistence. The main steps of the data pipeline are:

1. Data Acquisition
2. Batch Processing - Historical Data
3. Prediction Modeling
4. Real-time Stock Data Integration
5. Data Serving via Dashboards

In the project, we have setup an AWS EC2 instance with m2.2xlarge memory boosted instance type. We have chosen this instance type given we are performing batch processing in Pyspark and the heavy processing on the large historical dataset. The AMI is using the UCB EX2 image which comes with pre-installed Hadoop and Pyspark. In this project we are still using an single instance which supports both the data lake storage and batch processing power. We also tried with Auto Scaling Groups in AWS which in case the real-time data grows, the architecture is able to scale out without the need to relaunch a different instance type. However, when setting up and starting a Hadoop Cluster, Hadoop kept gives errors on the cluster configuration. It may requires launching new instances with fresh Linux images where Hadoop is not preconfigured to be using 1 instance.

The following diagram shows the high logical overview of the data processing pipeline:



Below are detailed steps on the data pipeline:

1. Data Acquisition: The raw datasets are acquired from WRDS server and loaded into data lake on Hadoop HDFS without going through any transformations. These datasets constitute the historical data which will be used to train the initial stock prediction model.

2. Batch Processing - Historical Data: After data is loaded to HDFS, the batch processing step will extract the raw data (csv files) from HDFS, apply schema when reading them, apply required transformations and clean the data by back/forward filling missing records. To speed up the batch process and also reduce data materialization during the data transformation, we have employed Pyspark throughout this phase. The output of the transformed data is stored as parquet tables back onto HDFS. This way the result can be persisted on the data lake for later usage. The detailed processing steps will be described in detail in the *Data Processing & Transformations* section.

   Additional to the batch data processing, we performed exploratory data analysis on the raw datasets. This helps us identify the format of the datasets and explore the features usable during the model building phase. Detailed steps and features found will be described in the *Exploratory Analysis* section below.

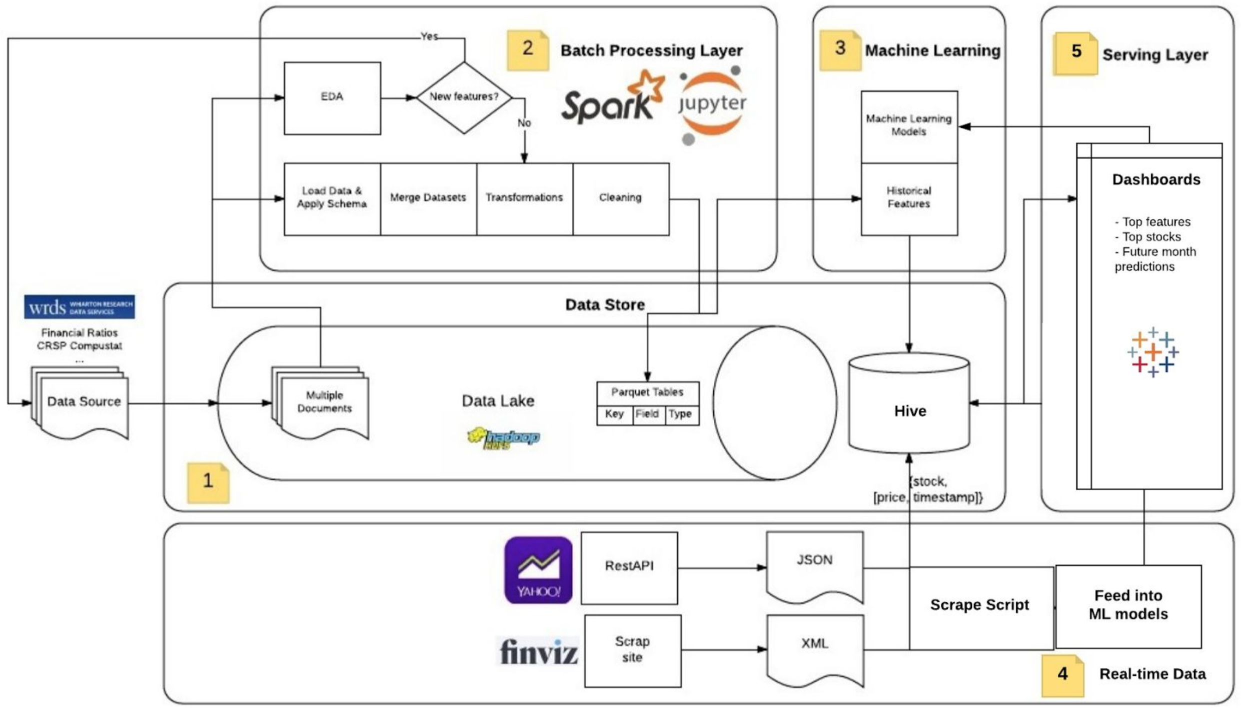
3. Prediction Modeling: The initial machine learning model is taking the historical dataset created by batch processing, training the model and getting the set of features that

predicts the stock performance best.

4. Real-time Stock Data Integration: The real-time stock data is access via a script which scrapes Finviz website on a daily basis. The features returned from scraping are then compared with features available in the prediction model. If a feature is identified as useful then it is incorporated to modify the parameters in the current prediction model. This step is to adjust the prediction model so that it will suit the most current stock prediction other than predicting based on data from months old. Technically a daily run of the script is still considered as batch data acquisition and processing to match the prediction model, this is the part of implementation that data is dynamic day by day and the prediction models adjusts according to the change.

5. Serving via Dashboards: After the daily stock data is incorporated, the prediction result is stored in a structured Hive table. We then serve the summary of the prediction result via Tableau dashboards connected by the Hive sql server. The dashboards presented are not representing the full set of features and aspects of the stock performance. Rather it is to show that by connecting to the Hive tables where all the prediction results are stored, users can dynamically create a new dashboard based on their interest. For instance, users can create a dashboard with only the set of stocks that contain dividend information.

The detailed architecture and data pipeline is shown as follows:



Historical Data Sources

Our approach towards data acquisition was extracting any and all stock market data that can be obtained without additional cost. Initially, our plan was to build a time series dataset by scraping data off of Yahoo! Finance, Finviz, and other financial screeners on a daily basis and adding the new rows to our historical dataset. However, after further discussion we found this approach to be problematic on multiple levels.

The primary obstacle we were faced with was the amount of time allotted for this project. Even if we started scraping financial data the day our group was formed the resulting dataset would only represent a few weeks which is limiting for a market as complex as the financial stock market and unlikely to provide substantive high level findings.

Secondly, the only primary keys we would be able to use from these sources are stock tickers. However, as we dug deeper we realized using stock tickers as our primary keys would pose problematic because they are not consistent over time. A company's ticker symbol becomes available for other's use when that company shuts down its operations, change its name, or is acquired. As a result, using the ticker as a primary key would record some changes in data over time for the same company while in fact it was two separate companies.

In an effort to mitigate these issues, we sought-after alternative options. Fortunately, we utilized UC-Berkeley's library database we have access to as students and found a multitude of resources with vast amounts of financial information. The one resource we found particularly helpful was Wharton Research Data Services (WRDS) which has dozens of different stock market datasets -- some of which have records dating back to the 1920s. Additionally, we found numerous unique identifiers intended for time series analysis. Each primary key represents one company despite any company name or ticker symbol changes.

After reviewing our available options, we opted to use the following (5) datasets:

1. **Financial Ratios Firm Level**
   Description: Monthly time series of financial ratios per security.
   Date Range: 01/01/1970 - 12/31/2015
   Columns: 75
   Rows: 2.6M
   Size: 1.4GB
   Feature Examples: Dividend Yield, Price / Earnings, Total Debt/Equity, etc

2. **CRSP/Compustat Merged Database - Security Monthly**
   Description: Monthly time series of stock performance.
   Date Range: 01/01/1962 - 12/31/2016
   Columns: 87
   Rows: 4.25M

Size: 2.6GB
Feature Examples: Monthly Closing Price, Cumulative Adjustment Factor, etc

3. **Beta Suite**
   Description: Monthly time series of regression data.
   Date Range: 01/01/1926 - 12/31/2015
   Columns: 13
   Rows: 2.5M
   Size: 0.6GB
   Feature Examples: Alpha, Beta, R-Squared, Idiosyncratic Volatility, etc

4. **Recommendations - Summary Statistics**
   Description: Monthly time series analyst recommendations from Thomson Reuters for each forecast period. Scaling is on a 5-point scale as follows:
   1 - Strong Buy
   2 - Buy
   3 - Hold
   4 - Underperform
   5 - Sell
   Date Range: 11/01/1993 - 09/31/2016
   Columns: 15
   Rows: 0.45M
   Size: 0.3GB
   Feature Examples: Mean Recommendation, Median Recommendation, etc

5. **CRSP/Compustat Merged Database - Linking Table**
   Description: Monthly time series of consolidated unique identifying information.
   Date Range: 07/01/1949 - Current
   Columns: 47
   Rows: 26,647
   Size: 12.3MB
   Feature Examples: PERMNO, GVKEY, CUSIP, TIC, etc

In summary, using WRDS as a resource dramatically increased the scope of work for our project. Originally, we hypothesized that in a best-case-scenario we would be able to scrape 15-25 days worth of data which would result in a dataset of around 80,000 rows across roughly 100 features. However, with WRDS we have access to a staggering amount of observations and features. The number of rows jumped from a scale of thousands to millions and number of features from hundred to hundreds.

Historical Data Acquisition

Additionally, WRDS provided access to datasets via a variety of methods. The first option we utilized was connecting to the WRDS cloud through the command line within a secure shell (SSH) which allowed us to copy datasets directly to our instance. This was our preferred method of acquisition since it opens up the opportunity to acquire this data through an executable script launched by a bash command which we can store within our EC2 instance and later apply a schema to. While this was an efficient method, unfortunately we couldn't acquire the *Financial Ratios Suite* dataset in this manner.

For example, some datasets were only available in SAS format which we could not read and analyze in PySpark. For SAS files, one of the way to overcome unprocessable data format is to convert the SAS file to text file using a python package called SAS7BDAT. The package allows reading in records line by line and write to a text file. However, when manipulating the output text file in Pyspark, parsing and applying schema is more complicated than csv data files. This is a complexity we had to dealt with. The second method is browser-based access where we could perform a web-based query and extract the data we need and in the format that we want. It allows specifying the stock features we are interested and also the time range of the query. From there, we downloaded the data onto our local machines and used secure copy (scp) to pushed it into our instance. Both of these methods include manual work, we were unable to totally automate the process. Since the WRDS website requires you to be logged in in order to access any data, we were unable to use *wget* commands streamline the data acquisition process. Alternatively, we tried setting up a script which copies raw data files from WRDS server with a password-less ssh. However, WRDS blocks such access thus restricts our automation possibility on data acquisition.

Exploratory Data Analysis

To get a better sense of the acquired data, we began merging and analyzing the datasets with the objective of informing what cleaning and transformations would be required to ensure unskewed high level observations. Additionally, we were searching for ways to enrich the data and potentially add informative features. Initially, the EDA was performed in Jupyter Notebook due our familiarity and that notebook can be followed at this link.

One of the first issues we noticed was that there wasn't a consistent primary key between the datasets. For example, the Financial Ratios Suite used GVKEY as a primary key while the CRSP/Compustat Merged dataset used PERMNO. Ultimately, this is what led us to incorporate the linking table dataset. However, after linking additional identifiers to a dataset we noticed duplicate rows. This is because there may have been multiple identifiers for a security and each were merged in resulting in duplicate rows. To counteract, we filtered any rows where the observation's date did not reside within the respective link dates supplied in the linking table (i.e., the timespan a security used a given unique identifier).

Additionally, since this was time series we had to ensure the datasets were merging not only with the correct stock, but also at correct time. As a result, we ended up creating time specific unique identifiers using the concatenation of a primary key with the current year and month (i.e., TIC-YY-MM, AAPL-17-04) which is ultimately what we ended up performing joins on.

The next key issue is that none of our datasets contained return over time -- our dependent variable. It's tempting to use a change in price over time as the basis for return, however this wouldn't be correct as it doesn't account for stock price splits or dividends. Fortunately, we had the variables to account for cumulative adjustment (splits) and total return (including dividends). Forward returns were calculated with the following formula:

$$FutureReturn = \frac{FuturePrice/(FutureCum.Adjustment \cdot FutureTotalReturnFactor)}{CurrentPrice/(CurrentCum.Adjustment \cdot CurrentTotalReturnFactor)}$$

The resulting value was our dependent variable -- total return that accounts for splits and dividends. Furthermore, using the same tactic past returns were calculated which acted as additional features.

Next, there were a few features which were categorical in nature. To provide numeric insight we converted each category of a feature into a separate feature with a binary output - 0 (false) or 1 (true). For example, if a security was within the Financial sector it outputs a value of 1 for the 'Financial_sector' feature and 0 for all sectors.

We wanted to enrich the data by adding two additional features for each continuous feature. For the first, we applied a modified Z-score with securities first grouped by each month. For the second, we applied a modified Z-score with securities first grouped by each month *and* sector.

In consideration of the timespan of this dataset, we wanted to added features that measure a security's standing per feature relative its respective sector or market on the observed date rather than historically because metric evaluations change over time. For example, what is an acceptable value for Price / Earnings value today may have been considered unfavorable 30 years ago. Furthermore, what is "normal" may also deviate across sectors.

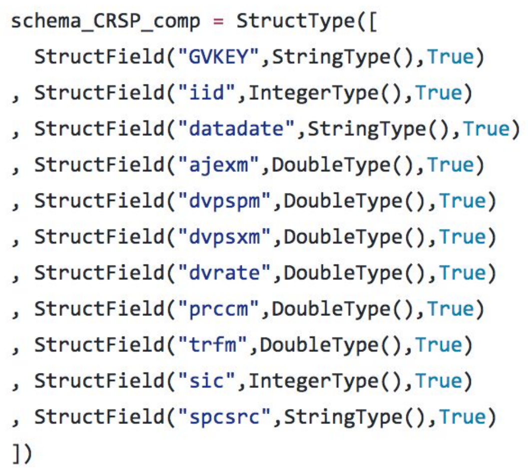
We opted for a modified Z-score in place of a standard Z-score due to the amount of outliers within the dataset. The modified Z-score is a more robust relative evaluation because it uses the median in place of the mean, and the median absolute deviation in place of the standard deviation -- unless the median absolute deviation is 0. In those scenarios the mean absolute deviation is used in its place.

To further mitigate outliers skewing the data we trimmed back the bottom and top 5% quantiles per each market month and again per each market month *and* sector. We preferred a clipping approach rather than removal because we wanted to retain the maximum number of observations.

Lastly, for machine learning algorithms to operate properly we needed to eliminate all null values. Due to the number of features, the nuclear options regarding nulls would leave us with very little data. What we found to be a better approach was forward filling and then back filling nulls by security, sector, and market whole in that respective order.

Historical Transformations

Throughout this phase, all the data processing has been done in Pyspark. We have chosen Pyspark on this phase since not only does it speeds up the processing in memory, but also that it reduces data materialization so that we are not storing unnecessary intermediate results in HDFS. The raw datasets that constitute the historical dataset used in machine learning phase are from WRDS monthly stock reports (Financial Ratios Suite, CRSP monthly security, Linking Table, Beta Suite and Recommendations). These datasets come in a CSV format in the HDFS Data Lake. As a first step, we read in the CSV files, apply schema on read that was identified during EDA and that store the data as parquet tables in HDFS. This step is to create initial structure on the datasets for later processing as dataframes in Pyspark. For fields with definite numeric meanings, we applied either float or integer type depending on the possible usage. For the stock keys and other unrecognized fields, we apply strings to them. Stock keys are applied string since they won’t be undergoing numeric operations and are supposed to be unique throughout the transformation and table merging. For each of the CSV data file, we store a separate parquet table in a separate HDFS directory. Examples of the application of schema and storing as parquet tables is shown below. To do this, we utilized the package spark-csv that supports applying of schema and creating dataframe in PySpark.



```
schema_CRSP_comp = StructType([
  StructField("GVKEY",StringType(),True)
, StructField("iid",IntegerType(),True)
, StructField("datadate",StringType(),True)
, StructField("ajexm",DoubleType(),True)
, StructField("dvpspm",DoubleType(),True)
, StructField("dvpsxm",DoubleType(),True)
, StructField("dvrate",DoubleType(),True)
, StructField("prccm",DoubleType(),True)
, StructField("trfm",DoubleType(),True)
, StructField("sic",IntegerType(),True)
, StructField("spcsrc",StringType(),True)
])
```

```
# READ IN FILES
financial_suite = sqlContext.read.format(sparkcsv).options(header='true').load(financial_suite_path, schema=schema_fin_suite)
crsp = sqlContext.read.format(sparkcsv).options(header='true').load(crsp_path, schema=schema_CRSP_comp)
link = sqlContext.read.format(sparkcsv).options(header='true').load(link_path, schema=schema_link_table)
beta = sqlContext.read.format(sparkcsv).options(header='true').load(beta_path, schema=schema_beta_suite)
recs = sqlContext.read.format(sparkcsv).options(header='true').load(recs_path, schema=schema_recs)

# EXPORT TO PARQUET
financial_suite.write.parquet("hdfs:///user/w205/financial_data/parquet_files/fin_suite")
crsp.write.parquet("hdfs:///user/w205/financial_data/parquet_files/crsp")
link.write.parquet("hdfs:///user/w205/financial_data/parquet_files/link_table")
beta.write.parquet("hdfs:///user/w205/financial_data/parquet_files/beta")
recs.write.parquet("hdfs:///user/w205/financial_data/parquet_files/recs")
```

After the schema is applied, we load the parquet into PySpark as dataframes and proceed with the transformations. The Financial Ratios Suite dataset is joined with the Linking Table dataset using the GVKEY. The Linking Table dataset contains links between different stock keys and thus are joined with the dataframe where we create a new unique key with "GVKEY-year-month" format. As a result, multiple datasets have a single record represents stock info reported per month. The new unique key guarantees that when datasets are joined, records belonging to the same stock and reporting month would be correctly matched. During the join, PySpark keeps the unique keys used for the matching, so it is necessary to drop identical columns so that there will be no duplicate columns in the dataframe. Below is an example on how dataframes can be joined in PySpark and also how the new unique keys are constructed.

```
df = fin_suite.join(link_table, fin_suite.gvkey == link_table.GVKEY, 'leftouter').drop(link_table.GVKEY).dropDuplicates()

df = df.withColumn('GVKEY_year_mth', concat(col('gvkey'), lit('-'), getYear(col('public_date')), lit('-'), getMonth(col('public_date')))) \
    .withColumn('CUSIP_year_mth', concat(col('cusip'), lit('-'), getYear(col('public_date')), lit('-'), getMonth(col('public_date')))) \
    .withColumn('TIC_year_mth', concat(col('tic'), lit('-'), getYear(col('public_date')), lit('-'), getMonth(col('public_date')))) \
    .withColumn('PERMNO_year_mth', concat(col('LPERMNO'), lit('-'), getYear(col('public_date')), lit('-'), getMonth(col('public_date'))))
```

After tables are merged, for each of the stock historical monthly records we add the past and forward 36 month stock features with regard to prccm, ajexm and trfm. To make sure when adding the past and forward stock levels, the records are not mixed up, we aggregate by the stock first and then sort the months for each stock. In PySpark, this is achieved by partitionBy command which places all records with the same GVKEY together, and then orderBy the new unique key GVKEY_year_mth we created. Thus, for each of the 36 past and forward months, we can just shift the records with the lag command and iterate through the rest to fill that it. Additionally, for each of the past and forward 36 months, the stock returns are calculated with the following equation

$$(\frac{(prccm_{past}/ajexm_{past}) \cdot trfm_{past}}{(prccm/ajexm) \cdot trfm} - 1) \cdot 100$$

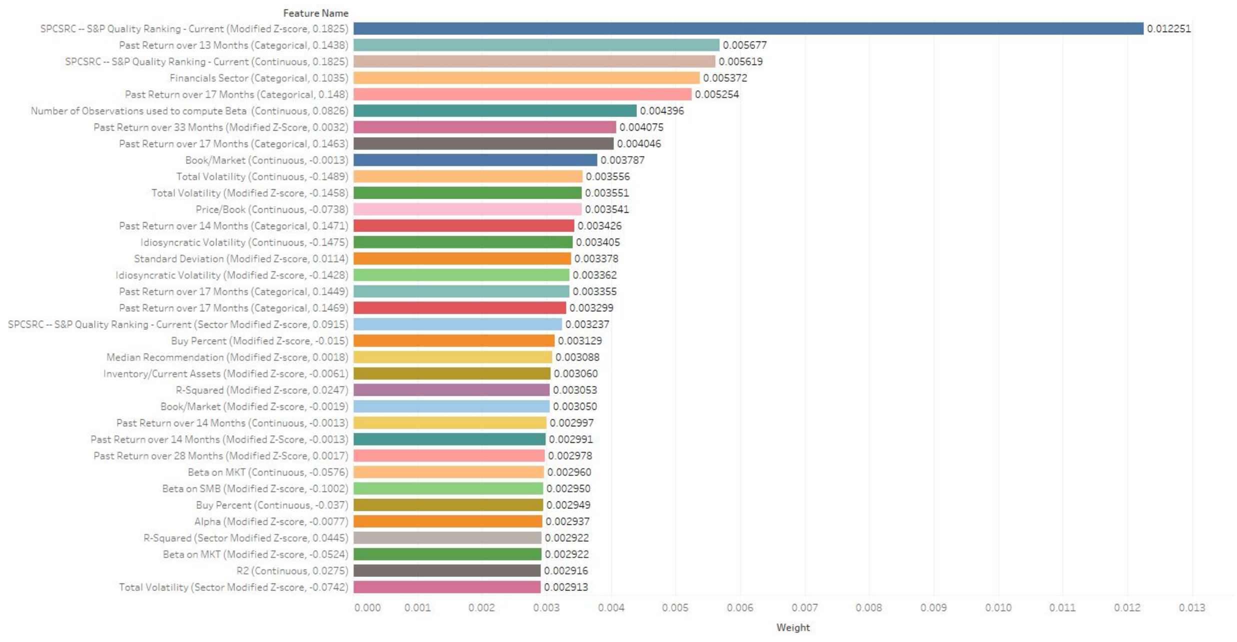
Below is an example function we used for adding the past and forward stock features.

```
def add_lead_lag(df, variable):
    for month in range(1,37,3):
        w = Window().partitionBy(col("GVKEY")).orderBy(col("GVKEY_year_mth"))
        first_new_col = "forward_"+str(month)+"_month_"+str(variable)
        second_new_col = "past_"+str(month)+"_month_"+str(variable)
        df = df.withColumn(first_new_col, lag(col(variable),-month,None).over(w)) \
        .withColumn(second_new_col, lag(col(variable),month,None).over(w))
    return df
```

Feature Analysis

Using the cleaned historic data, we are able to analyze feature relationships to return on investment (the following notebook can be followed for our process and findings). As a default, we arbitrarily chose forward return over the course of 12-months are the dependent variable with a binary output -- 0 for a loss on investment and 1 for a profit on investment. Using a binary categorical dependent variable seemed to reduce noise and increase the magnitude of correlation between variables.

To analyze feature importance, we calculated correlation with 12-month return and each feature using Pearson's method. Additionally, we used the Scikit feature selection module, ExtraTreesClassifier, which ranks each feature importance and outputs a weight for each. Below is a visualization which is sorted by the weight assigned to each feature based on importance and the correlation is listed after the feature name.



The factor with the largest weight is the S&P Quality Rating score. However, after further research the value given is *today's* current grade assignment so this is an incredibly biased variable and should be ignored.

Outside of that, the biggest takeaway from this analysis is the importance of past performance as an indicator of future importance. Stocks which returned a profit tended to continue to do so in the future.

The next feature that comes with largest correlation is volatility metrics. The more volatile securities tended to have a negative correlation with profitable return.

Dividends also were also one of the more correlated features which makes sense since that is a more secure return than change in stock price.

Of the sectors, the Financial sector held the highest correlation (positive). The next closest in magnitude was the Information Technology sector, however the correlation was negative.

Lastly, there is a negative correlation regarding analyst recommendation and actual outcome -- at least for 12 months out. When an analyst suggests a sell, statistically speaking that's when an investor is best buying and vice versa regarding an analyst suggesting a buy.

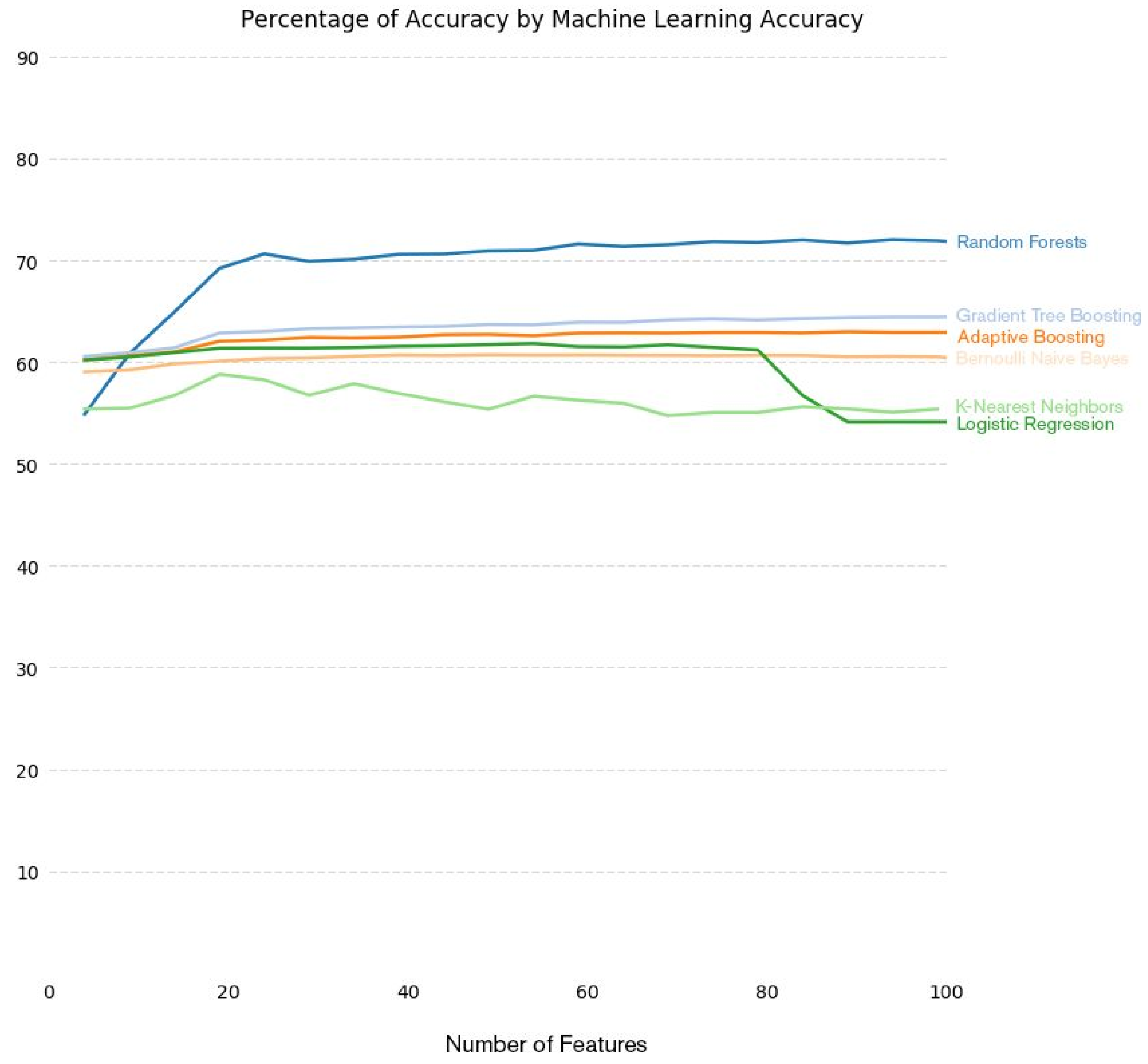
Model Analysis

Initially, our aim was to predict both probability of a return or loss, a classification problem, and magnitude of a return or loss -- a regression problem. To understand what machine learning algorithms were most appropriate for each, we sampled 25% of our data which was then split into 70% training and 30% test data.

For classification, probability of a return or loss, we used the following algorithms:

- Logistic Regression
- Bernoulli Naive Bayes
- Random Forests
- Adaptive Boosting
- Gradient Tree Boosting
- K-Nearest Neighbors

We tracked accuracy for each algorithm by iterating through a loop that adds a new feature in the order of the earlier ranked feature importance. The results can be viewed in the visualization below:



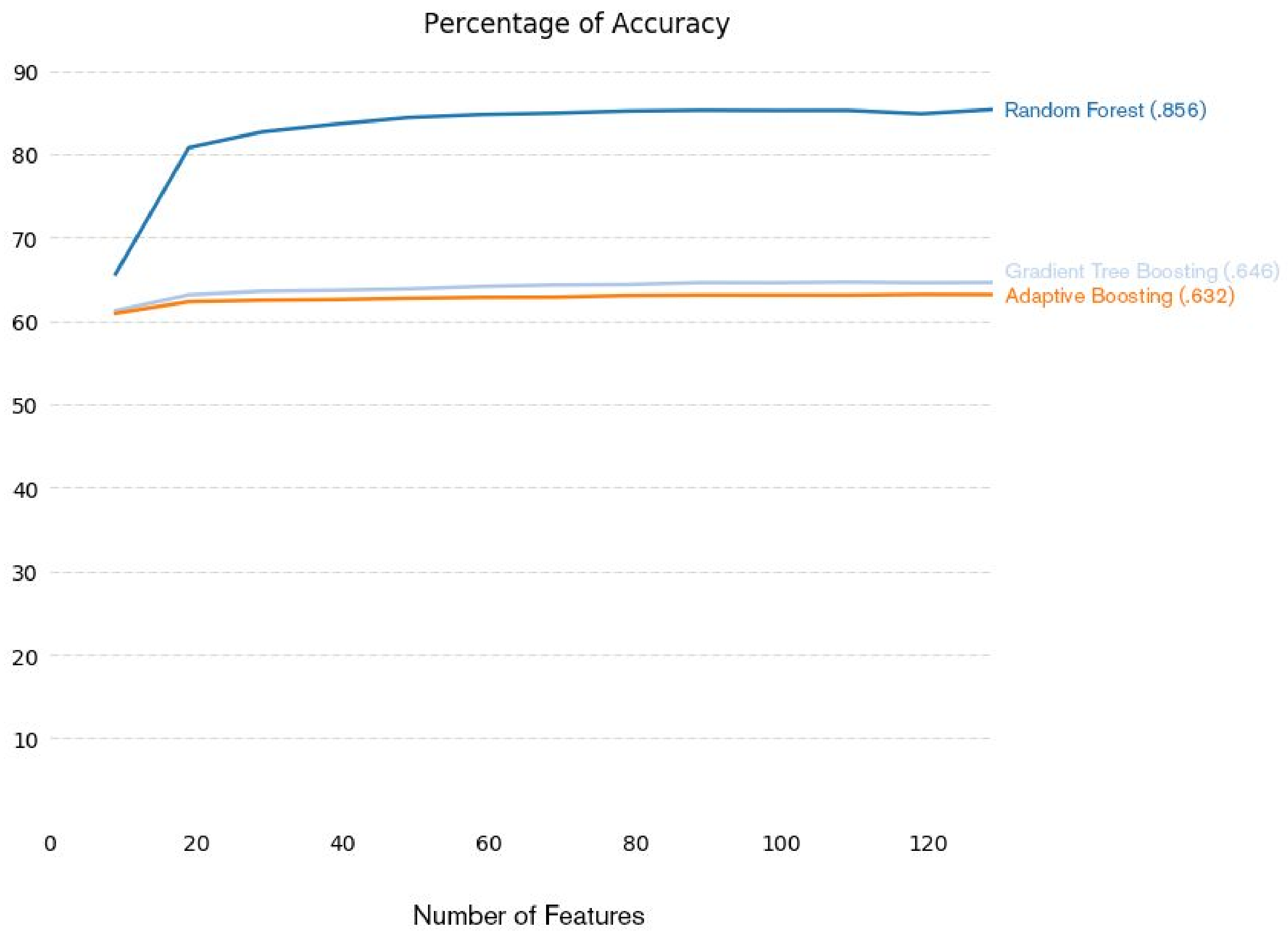
Initially, we hypothesized a 0.60 accuracy rate is a successful project outcome based on similar research projects done by others and the complexity of the market. The initial results we received seemed to be in line with that result. However, the Random Forests algorithm continued to improve as additional features were added -- eventually surpassing 0.70 accuracy.

Unfortunately, our regression results did not return the same success -- in fact their R2 score was in the negatives and MSE astronomical.

We ended up focusing our efforts towards the classification problem and moved forward with the top three performing algorithms, Random Forest, Adaptive Boosting, and Gradient Tree Boosting, and tested each on the complete dataset across over a hundred features. Again, the data was split into a 70 to 30 ratio of training to test data and recorded the respective accuracy, higher being better, and log loss, where lower is better. The results are as follows:

Random Forests
Accuracy: 0.856
Log Loss: 0.522
Adaptive Boosting
Accuracy: 0.632
Log Loss: 0.691

Gradient Tree Boosting
Accuracy: 0.646
Log Loss: 0.628



With the larger dataset, the accuracy did improve though Adaptive Boosting and Gradient Tree Boosting seemed to plateau around 0.64 accuracy. However, Random Forest continued to improve to a final accuracy of 0.856.

As stated earlier, our expectation was around a 0.60 accuracy so our 0.86 result was a pleasant surprise.

Real-time Data Sources & Acquisition

With the historic data cleaned, analyzed, and available for training a predictive model, our next task is to acquire real-time stock data using the same features our model is trained on. Unfortunately we're not able to use the same sources of data that were utilized for our historic data.

However, our previous strategy of scraping financial data works well in this case. We only need the current day's data and the ticker symbol can be used as a primary key since it isn't time series data -- each ticker symbol represents only one company.

Accordingly, we built a script to scrape data off of Finviz, a stock screener, and Yahoo! Finance. Finviz's financial data originates from Reuters and stored on Finviz in a retrievable XML format using Python's 'BeautifulSoup' package. This is not only an excellent source for financial data, it is also a consolidated listing of stock tickers from several stock markets (NYSE, Nasdaq, etc).

Using the tickers we retrieved from Finviz, we query Yahoo's query language for additional financial information. For each feature on Yahoo! Finance, the script iterates through queries with the only difference being the stock ticker symbol. The result is an output for each individual security which is stored in a JSON format.

Real-time Transformations

The real-time data required less cleaning than the historic data since we were not training data off of it so outliers skewed the performance less. However, we need to ensure the format more or less matched the historic dataset. The same features, cleans, and transformations.

Due to the source data (Yahoo & Finviz), null data wasn't recorded as a NaN. Instead it's shown as a " - ". So the first step was to convert all " - " to a NaN which would later be filled.

The next step was to strip all " % " symbols across all features and convert the data to numeric instead of a string value.

Following that, the current month (of extraction date) is recorded and entered in the respective "month" column to match the binary output in the historic section. 'Sector' also required binarization as well, however the naming of sectors did not match the historic dataset. To compensate, we researched each respective sector and associated industry to help create the closest sector matches between the historic and real-time dataset. The last binarization was past performance -- companies posting a profitable return over a given period of time received a ‘1’ while unprofitable returns receive a 0.

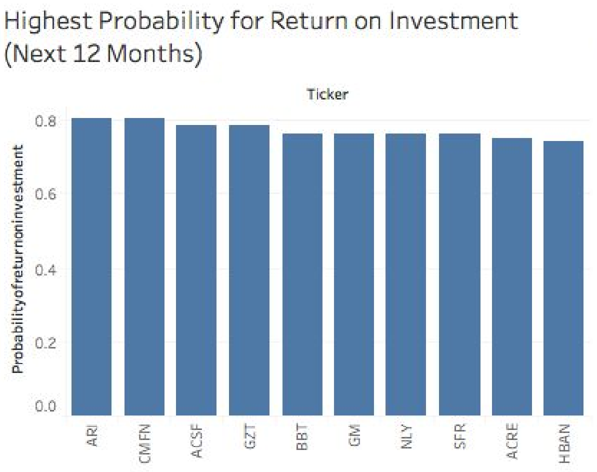
To ensure the dataset was formatted correctly for the machine learning algorithm, null values were replaced with the median of each feature.

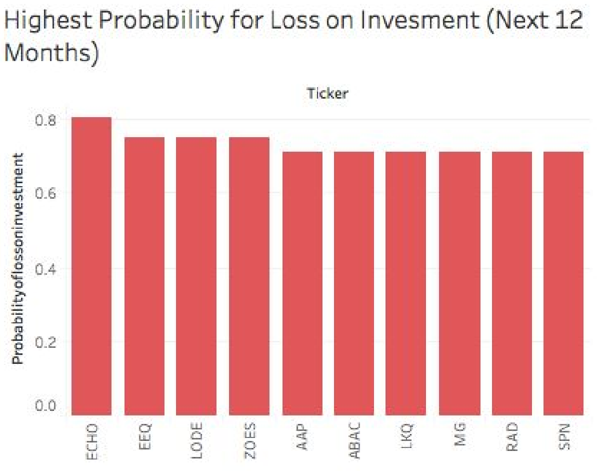
The modified Z-score was applied to the continuous features that were in both the historic and real-time dataset.

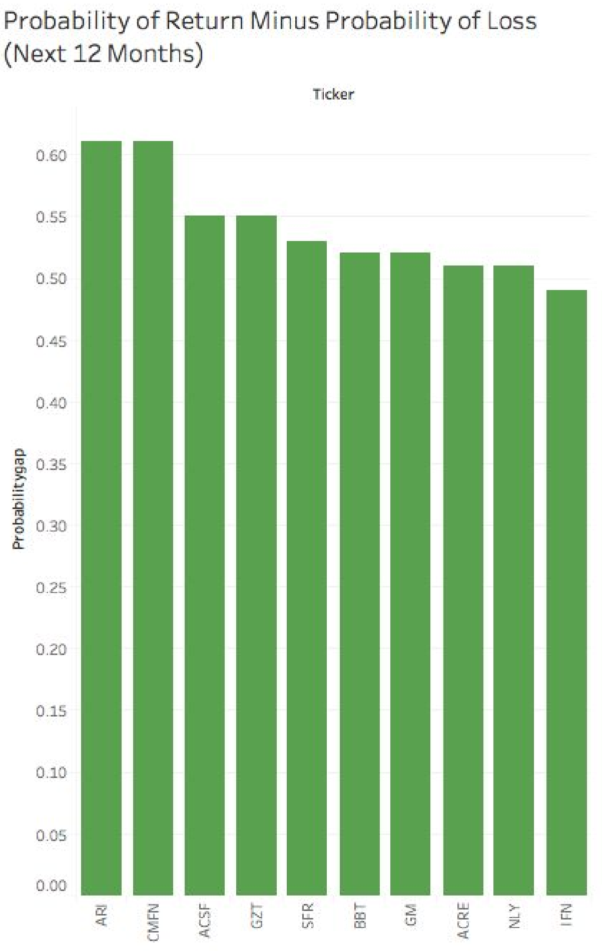
Lastly, we renamed the common features to match the historic dataset and applied the machine learning model that was trained from our historic dataset. With that we were able to add the probability of either a profitable or unprofitable return.

Stock Prediction Results

Once we created the stock predictions, we wanted to be able to visualize the results and make them easy to present. To accomplish this task, we decided to use Tableau. First, we created a Hive table with our predictions data, then we connected Tableau to Hadoop. Below are three charts in Tableau showing the top ten stocks for each of the three types of predictions we made. The Hive table contains many other variables, but these are the ones of main interest.







Technical Challenges

Our first challenge was finding a good historical data source. We initially thought we would have to build it ourselves because it's difficult to find free sources that will include historical stock prices. After a lot of digging, we found WRDS which solved this issue for us.

Once we solved that issue, we ran into trouble with the size of the datasets. The full datasets were too big for Pandas to handle and while Spark dataframes could handle them, we had issues with Spark dataframes as well (more on that later). In order to move forward with exploratory analysis and not be held up by Pandas' size-limit or Spark dataframe issues, we decided to subset the data and use Pandas to work on a smaller portion of data.

The next challenge was installing Spark 2.1. We planned on using the sklearn and scilearn machine learning tools and they require Spark 2.0 or higher. The pre-built UCB MIDS images contain Spark 1.5 so we needed to install the newer version. The instance was set up with everything designed to use Spark 1.5 so trying to reconfigure it to use Spark 2.1 instead was time consuming. Even after we thought everything was properly configured to run off Spark 2.1, there would still be processes erroring out by trying to reference Spark 1.5.

Finally, what caused us the most difficulty was all of the work we did in Spark, specifically with Spark dataframes. None of us have experience with Spark dataframes, and they are not as easy to work with as Pandas or other dataframe tools in other languages. For every simple command in Pandas, there was a lot of effort to translating it into Spark dataframes. The coding isn't as simple or neat and the online resources are minimal. We spent a lot of time on Stack Overflow trying to learn how to work with Spark dataframes.

For specific issues, first, we had trouble reading data in correctly and getting into a usable format where we could apply the schema to it. For this reason, we used solely CSV files because the pipe delimited text files were giving us too much trouble. Second, we wanted to calculate forward and past returns for each row. Figuring out how to create new columns where we essentially lagged values from another column took us a couple weeks. Third, we spent over a week trying to figure out back-filling and forward-filling the null values for each column and in the end weren't able to get it to work. We successfully ran code to do back-filling and forward-filling on a test dataset, but when trying to apply it to our actual dataset, we ran into errors every time and could not figure out the source of the issue. Fourth, we had numerous memory issues when trying to run transformations in Spark. Since Spark uses lazy evaluation, we input all the transformations we wanted, but then we when actually went to call the data, such as save the output to a file, Spark would finally compile it and then crash. After struggling with this for a while, we solved it by changing the Spark local directories to be on the */data* where we had more memory and storage. However, that only solved our memory issue when launching PySpark and running the code line-by-line. When trying to run an entire file using *spark-submit* we continued to run into memory errors and were not able to run the code successfully. Even small steps along the way, such as adding a column, merging datasets, filtering, took significant time to learn.

Eventually we tried doing the heavy computations in PySpark dataframes and then converting the dataset over to Pandas to perform any additional steps. We were unable to accomplish this task.

Limitations and Future Extensions

Our project has several limitations. First, while we found rich historical data to use, we were unable to find a real-time data source with lots of data. The real-time data we scraped had much more limited features than the historical dataset. This limits what variables we can include in our machine learning model, since we must only include variables that can be applied to the real-time data. However, if there comes a point in the future where more features are accessible from scraping or there is another way to get better real-time data, we will be able to apply a more complex model since we have the historical data loaded already.

Our next limitation is that the loading of historical data was a manual process. We needed to sign into the WRDS platform, perform a web-based query, download the results to a local machine, copy to our instance, unzip the files, and then move it HDFS. We were unable to find a way to automate this process. While doing this manually was fine for our project, it limits the usefulness going forward. Ideally, once WRDS gets updated with more data over time, we would like to automate the ingestion of data to HDFS. Several of the other processes we did along the way took manual work as well. In the future, we would want to clean up the code to make things run more smoothly and automated from end to end.

Resources & References

[1] Project GitHub Repository https://github.com/jeffrey-hsu/W205_Project

[2] Wharton Research Data Services https://wrds-web.wharton.upenn.edu/wrds/

[3] Finviz Financial Visualizations http://finviz.com/

[4] Yahoo! Stock Most Active https://finance.yahoo.com/most-active/