May 9, 20201

Predicting the Stock Market

Big Data Final Project Report

Taylor Rohrich (trr2as) Alex Stern (acs4wq) Chirag Kulkarni (ck3fz)

1.0 Abstract

Through analyzing data from the New York Stock Exchange (NYSE), our team intends to evaluate a variety of models for exploring the research question of “can we predict the forecasted price of stocks one month from the current date?”. We convert the problem to a binary classification problem by creating a response that is ‘1’ if the stock price improves in one month, and ‘0’ otherwise. Throughout this paper we will outline our data exploration and preprocessing steps, as well as the creation of a reproducible pipeline in spark that is generic and can be utilized for multiple models. We will explore a logistic regression model as a baseline, and extend this research by creating both a multilayer perceptron neural network and a gradient boosted trees model. After creating these models and tuning the hyperparameters via a train validation split approach, the gradient boosted tree model proved to have the best results, being able to effectively predict if a stock will improve in the next month with a test set accuracy of 77.9%, while the other two models failed to predict anything but the positive class.

2.0 Data and Methods

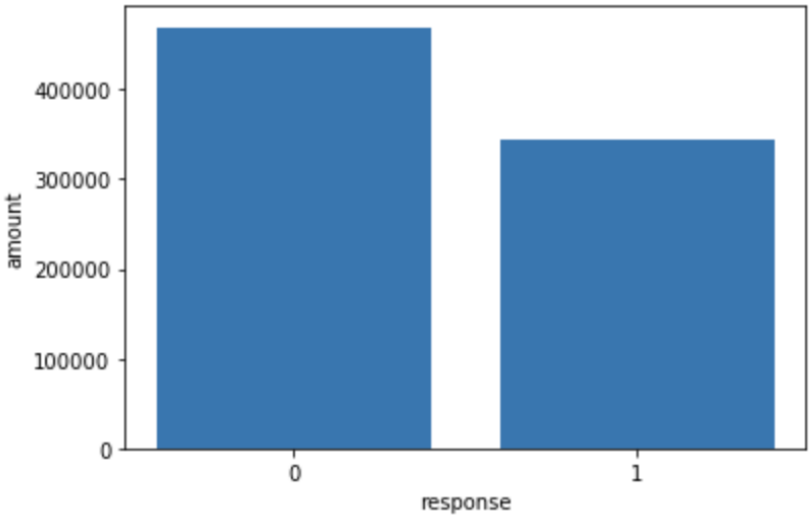
2.1 Data Introduction

There are two main methodologies to stock-price forecasting: technical analysis and fundamental analysis. Technical analysis disregards specific information related to the company; instead, it focuses on stock-price movement and if a pattern suggests the stock will move in a certain way. Fundamental analysis looks at indicators about the company’s health, such as liabilities, asset ratios, etc. to evaluate the “true-value” of a stock and determine if the current price is under or over-valued. For our project, we decided to conduct a hybrid analysis that draws from both fundamental and technical analyses.

We decided to use New York Stock Exchange (NYSE) data sourced from Kaggle here. The “prices” dataset contained four features that we used: date, stock ticker

(*symbol*), the open price of the day (*open*), the closing price of the day (*close*). We split the date into the day of week, day of month, year, and month. This was to account for any regular temporal movements, e.g. retail stocks do better around the holiday season. We pulled in gross-profit and long-term debt from the “fundamentals” dataset to include a snapshot of company-specific value.

After we decided on our feature set, we still had to engineer our response variable, future-price. As a group, we decided to forecast prices out by 1 month. We found the forward price (4 weeks ahead of the current observation) and appended that to each of our records with a high hit rate of about 96%. The missing 4% were due to the forward query falling on weekends or holidays; the dataset still maintained its integrity, though, so we dropped those values and proceeded with model building. We decided to use a 70-30 train-test split for our data and decided against down / up-sampling because the two categories of the response were approximately balanced as shown in the figure below:



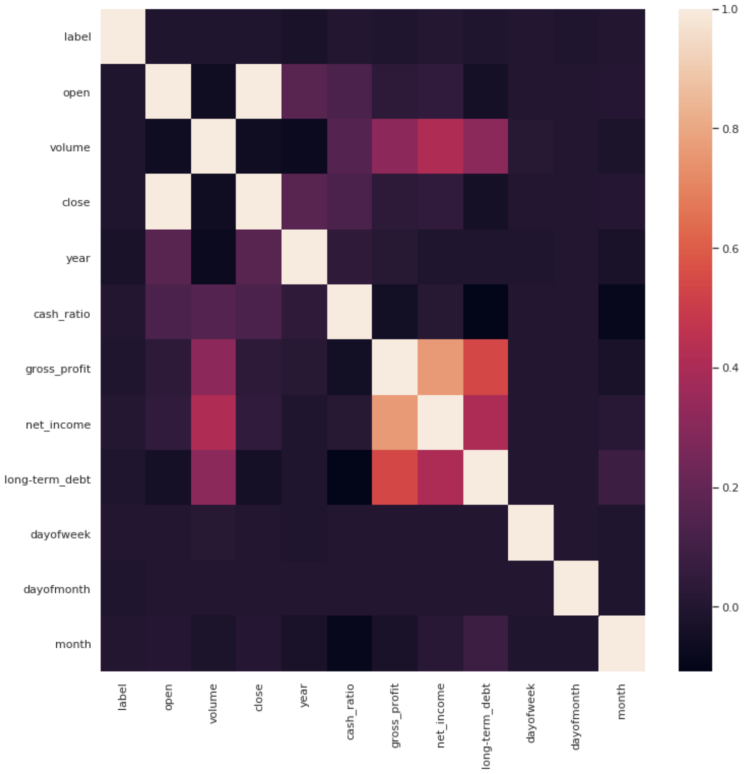
2.2 Preprocessing Methodology

To implement PySpark best practices, our data ingestion process for our model training consists of utilizing pipelines for reusability and error handling. Due to the financial nature of our data, there are several incremental preprocessing steps that are required in our pipeline. The first preprocessing step is to change the data type of specific columns. For example, due to the timeseries nature of our data, the ‘date’ data type is incompatible for several machine learning model operations, so we transform date columns to integers. Secondly, we impute NA’s for the columns in our dataframe utilizing the Imputer transformer. Third, we convert categorical variables to an index --

this is especially important for our “Ticker” column which has 501 unique values, as demonstrated in the output from our exploratory data analysis below:

```
C       1762
AIG     1762
PEG     1762
PHM     1762
IDXX    1762
        ...
KHC      378
HPE      304
CSRA     284
WLTW     251
FTV      126
Name: symbol, Length: 501, dtype: int64
```

We then one-hot encode these indices utilizing the OneHotEncoder Transformer. Next, we had to encode our response variable. We have converted our problem from a regression problem to a classification problem of ‘if a stock will be up or down from its current value in 1 month’. To create this label, we subtract the price of a specific stock 1 month from now from the current closing price, then convert this result to either a zero or a one if it is greater or less than zero. The final preprocessing step in our pipeline is to select the feature we want to utilize in our models from the entire dataset of more than a hundred columns. We utilize the VectorAssembler transformer to create the selected ‘features’ column. Feature selection was driven in part by the correlation matrix created during our exploratory data analysis:



From the output above, we see that there are not a lot of predictors that have significantly high correlation with the response (label), yet there are several predictors that highly correlate with each other, Thus, we chose to pursue predictors that were both relatively high in correlation with the label but also not strongly correlated with each other. We thus ended up with the following predictors: ‘'open','close', 'year', 'dayofweek', 'dayofmonth', 'month','date' ,'symbolIndex', 'gross_profit','long-term_debt', and ‘'close', dropping ‘net_income’ because it had a fairly high correlation with ‘gross_profit’.

One unique thing to note about our preprocessing steps is that we had to utilize custom Transformers for several parts because PySpark prebuilt Transformers could not accomplish specific tasks. These tasks included: automatically changing column data types and creating a response variable. Although the workload spent on this preprocessing pipeline was extensive, it nevertheless set us up to have a scalable, error-resilient pipeline that could be reused regardless of model type as we began training models discussed in the sections below. Indeed, shown in the example pipeline below, we can easily reuse all of the stages of the pipeline, save for the final step which is the model itself:

```
pipeline = Pipeline(stages=[changeColumnType,imputer,indexer,responseVariable,encoder,getFeatures,gbt])
```

2.3 Model Methodology

The first model we will train is a basic Logistic Regression model. First developed in the 19th century, the logistic model was the very first model trained for a binary (probabilistic) output. In our model, all of the features are linear. This means that we assume none of them can be better represented as quadratic, cubic, etc. terms. We declined to include interaction terms in our feature space on the basis that domain knowledge suggests this would not be prudent given the features we selected to use across our models. This simple model has two hyperparameters that we were able to tune: the maximum number of iterations (*maxIter*) and the value of the elastic net parameter (*elasticNetParam*). The elastic net determines what level of mixture between lasso and ridge penalties our model will include. These are both regularizing constraints on the model. Ridge acts as a weighting factor, shrinking the contributing effect of some features while increasing others. Lasso acts as a feature selection tool, shrinking the contributing effect of some features to zero (ridge can only shrink to non-zero weights).

Our second model, the deep learning model, contained four layers. The layers contained eleven, five, four, and two nodes, respectively. The final layer of this type of

model contains a single sigmoid node. This sigmoid function performs the same as a traditional logistic regression model does, outputting a probability that the observation belongs to the positive class. The probability that it belongs to the negative class is thus one minus this value. Our deep learning model had two hyperparameters we needed to tune: the number of epochs the model is trained for (*maxIter*) and the size of the batches fed into the model during each epoch of training (*blockSize*). An epoch is an iteration of running the full training dataset through the deep learning model and updating the weights of each layer. A batch is the number of training examples fed into each run through the model. Once all batches have been run through, an epoch has been completed. These hyperparameters can change how quickly or slowly the deep learning model updates itself and are important to tune.

Our final model is a gradient boosted tree model. This is an extremely powerful model and has become the industry standard for a variety of tasks across a plethora of industries. There are two hyperparameters we tuned: the maximum number of bins used for splitting features (*maxBins*) and the maximum depth of any individual tree in the model (*maxDepth*). The number of bins is always greater than or equal to the number of features that each individual tree could possibly use. The depth of the tree ensures that no individual tree overfits and allows the gradient boosting algorithm to improve its classification ability properly.

When considering the hyperparameter tuning procedure for the models, we looked at both a k-folds cross validation approach as well as a train-validation split approach. Due to the extremely computationally intensive nature of our models (especially the neural network) and the shear size of our dataset, we decided to go with the train-validation split approach to hyperparameter fine-tuning. Although this approach may not be as accurate, it allowed us to run our hyperparameter fine-tuning procedure in a reasonable amount of time.

3.0 Results

After outlining our methodology concerning pipelines and hyperparameter tuning above, we can now explore the results of our three types of models, and our resulting champion model.

3.1 Logistic Regression

The logistic regression model created can serve as a ‘benchmark’ model, being the least complex of the three models created. The hyperparameter to be tuned for

Logistic Regression included the max number of iterations as well as the elastic net parameter. After the hyperparameter tuning method outlined above, we received the following results for the Logistic Regression Model:

	areaUnderROC	maxIter	elasticNetParam
0	0.517932	10	0.1
1	0.517932	10	0.5
2	0.517932	10	0.9
3	0.517905	20	0.1
4	0.517905	20	0.5
5	0.517905	20	0.9
6	0.518338	30	0.1
7	0.518338	30	0.5
8	0.518338	30	0.9

From the results above, we see that all of the results are very similar, but the best hyperparameters are 30 iterations and any value of the elastic net parameter: 0.1, 0.5, or 0.9, as all have a AUC value of 0.5183 (which is barely better than guessing). With these optimal values of the hyperparameters, we proceeded to evaluate our model on the test set and report the following metrics:

	auc	accuracy	precision	recall	f1score
0	0.5	0.575036	0.575036	1.0	0.419884

	predicted 0	predicted 1
actual 0	0.0	104067.0
actual 1	0.0	140817.0

From the output above, we can highlight that the accuracy is 57%, and from the confusion matrix we see that the model is classifying every stock as improving! As far as an intelligent classifying model, this is poor performance, but nevertheless this is the benchmark model.

3.2 Deep Learning Model

Next, we can analyze the hyperparameter tuning of our deep learning model. For this model we were interested in looking at the hyperparameters of the number of iterations as well as the block size. The output is shown in the table below:

	areaUnderROC	maxIter	blockSize
0	0.503215	10	64
1	0.503215	10	128
2	0.503215	10	256
3	0.503215	50	64
4	0.503215	50	128
5	0.503214	50	256
6	0.503215	100	64
7	0.503215	100	128
8	0.503214	100	256
9	0.503215	150	64
10	0.503215	150	128
11	0.503214	150	256

The results show that the hyperparameter choice makes no difference: all produce an AUC of about 0.50: essentially guessing. This is especially surprising because deep learning models traditionally work well with representing nonlinearities; nevertheless we will now look at the evaluation of this model with metrics on the test set prediction, utilizing 10 iterations and a block size of 64.

	auc	accuracy	precision	recall	f1score
0	0.5	0.575036	0.575036	1.0	0.419884

	predicted 0	predicted 1
actual 0	0.0	104067.0
actual 1	0.0	140817.0

The result above is identical to that of the logistic regression model: this model is essentially guessing via the 0.5 AUC score, as well as classifying all of the examples for stocks of specific dates as improving for 1 months for now. Both these models are essentially useless-- we finally turn to the gradient boosted tree model below.

3.3 Gradient Boosted Tree

With the Gradient Boosted Tree, we were interested in analyzing the hyperparameters of the number of max bins as well as the max depth of the trees. When running our hyperparameter tuning methodology explained above we get the following results:

	areaUnderROC	maxBins	maxDepth
0	0.731846	550	5
1	0.845648	550	10
2	0.738461	700	5
3	0.831500	700	10

At first glance we finally have a model that performs better than guessing: the best hyperparameters appear to be 550 bins and a max depth of 10. The corresponding AUC is 0.845, which is significantly better than guessing. After retraining this model on the full train set, we evaluate it on the 30% test split and receive the following metrics:

	auc	accuracy	precision	recall	f1score
0	0.759694	0.779144	0.764874	0.889303	0.773794

	predicted 0	predicted 1
actual 0	65571.0	38496.0
actual 1	15588.0	125229.0

From the output above, we see that our Gradient Boosted Tree model performs the best on the test data of the three, with a test set accuracy of 77.9%. Looking at the Confusion Matrix, we do see that our model does have a higher rate of false positives than false negatives, with a false positive rate of about 37%. Nevertheless, stock market data is traditionally a domain where extremely high accuracy is uncommon -- this Gradient Boosted Tree model performs the best of our three models, boasting an overall test set accuracy of 77.9% and thus is our champion model.

3.4 Sensitivity Analysis

We conducted brief sensitivity analysis into the resulting optimal hyperparameter models created for the three types of models above. We changed several of the predictor variables by one standard deviation and analyzed how the resulting test set accuracy changed, the results are below:

	LR	DL	GBT
gross_profit	0.576032	0.576032	0.781389
long-term_debt	0.576032	0.576032	0.781728
open	0.576032	0.576032	0.781058
close	0.576032	0.576032	0.781585

The values in the table above are the test set accuracies by modifying a specific variable by one standard deviation. Compared with the test set accuracies shown above, all of these deviate from the non-adjusted predictors by less than one percent, suggesting that our models are robust to perturbation.

4.0 Conclusions & Next Steps

We are very happy with the champion model we were able to produce, being able to effectively predict if a stock will improve within one month with 77.9% accuracy. The test accuracy of the model gives us optimism about its ability to predict the future price movement of an unseen stock. We believe this is just the start of what could be a much broader, comprehensive hybrid approach to predicting future movement of the stock market. A robust approach relying on cross validation could likely yield a higher test accuracy. As well, we hypothesize that a model including a time series-based approach to future prediction could better incorporate previous information about the movement of a stock's price. Additionally, creating industry-specific models could yield better results if certain industries are easier/harder to predict or follow certain trends. Finally, we speculate that text analysis could lead to the engineering of useful features for the prediction of future stock price movement.