An Application of Natural Language Processing to Stock Forecasting

Nicholas Bayne Grubb

April 2020

Introduction

The question posed in this project is simple. Can a Machine Learning Algorithm predict the rise or fall of the stock market based on the news from a certain day? It is a known fact that the stock market reacts to the events transpiring in the world around it. So, given a wealth of news reporting on those events, can we train an algorithm to predict how the market will react to those events?

The Dataset

The dataset was gathered via web scraping from the The Wayback Machine [9] for pages from Bloomberg [3], Barron's [1], and The Wall Street Journal [8]. Urls from January 1st, 2015 to April 6th, 2020 were gathered and the text of the page was saved using Selenium. The dataset was saved into 135 .csv files, which were then combined and cleaned using `combine_csvs.py` and `clean_data.py`. The last 10 years of history of the S&P 500 was then gathered and labeled using `harvest_stonks_data.py`. These labels are the targets of our binary classification problem. The labels are `True`, indicating that the S&P 500 fell in value during that day's trading, and `False` indicating that the value rose on that day. Each data point in the dataset is a day, with all of its associated news articles from every website from `bigboi.csv`, labeled with it's respective day from `labeled_spy_data.csv`.

Data Pre-processing

Though it went through an initial stage of cleaning, the data is not able to be used by a Machine Learning algorithm yet, and requires more pre-processing. The data was pre-processed using a CountVectorizer from Scikit Learn [5] to implement a Bag Of Words approach, and TextBlob [7] and NLTK [2] to perform Tokenization, Stop Word removal, Part of Speech Tagging, and Stemming. More information the text preparation can be found in the implementation on lines 14 and 84 of `machine_learn.py`.

The Algorithms

The algorithms used for this project were Scikit Learn's Multi-Layer Perceptron (MLP) Classifier, Scikit Learn's Support Vector Machines (SVM) [5], and eXtreme Gradient Boosting (XGB)[11].

Multi-Layer Perceptron

This is Scikit Learn's implementation of a Neural Network. The parameters tuned were `alpha` and `hidden_layer_sizes`. The hidden layer parameter was, in this case, 4 different inputs. Each input represented the size of one of the four hidden layers in the Neural Network. The regularization term `alpha` reduces the complexity of

the hypothesis class. Larger values for `alpha` mean less complex decision boundaries are learned by our classifier[10].

C-Support Vector Classifier

This is Scikit Learn's implementation of a Support Vector Machine Classifier, using a Radial Basis Function kernel. The parameters tuned were `C` and `gamma`. `C`, the regularization term, again allows for less complex decision boundaries when its value is high. Thinking about this in terms of the Large Margin Principle that is the basis for SVM's operation, a lower value of `C` means a larger margin between the cost of training classification accuracy, while a higher `C` means higher training classification accuracy with a smaller margin. Gamma defines how far the influence of a single training example extends, with a higher value allowing for lower influence for a training example [6].

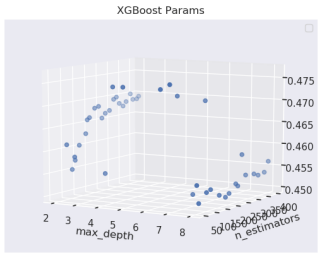
eXtreme Gradient Boosting

eXtreme Gradient Boosting (XGB) is a highly optimized gradient boosting library. It was used in this project to implement decision tree based gradient boosting. The algorithm works by fitting many different weak learning decision trees to a dataset. The parameters tuned were `max_depth` and `n_estimators`. `max_depth` limits the maximum depth of each weak tree fit by the dataset. More depth allows for a more complex hypothesis to be learned by each weak leaner. `n_estimators` is the number of weak learners to fit, with more trees being fit also allowing a more complex decision boundary.

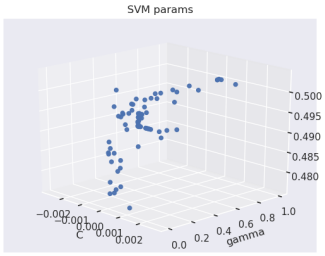
Hyperparameter Tuning with Bayesian Optimization

The hyperparameters of these algorithms were tuned using Bayesian Optimization [4]. In short, this optimization method works by constructing a posterior probability distribution on the function it is optimizing, then targeting regions that are more likely to have higher values based on this probability distribution. The functions being optimized in this case were the `roc_auc` scores of a 5-fold Cross Validation split. This was implemented via Scikit Learn's `cross_val_score` using Scikit Learn's `TimeSeriesSplit` to implement a version of Roll Forward Cross Validation in which the training data grows with each iteration[5]. The Bayesian Optimization was run with Fernando Nogueira's library [4]. 45 optimization iterations were run on each algorithm for a total of 23 hours of computation time between the three algorithms tuning. For SVM, an additional 45 iterations were run to explore an edge of the parameter space not covered by the previous tuning that seemed to produce good results. The below graphs of each algorithms paramaters vs. score were usde to aid in the selection of the final models. In each graph, the vertical axis is the mean ROC-AUC score for each iteration of optimization with those parameters. For more information, see the implementation at `baynesian_optimization.py`, and the results at `svm_results.csv`, `xgb_results.csv`, `mlp_results.csv`. The graphs below were generated using `plot_cv_results.py`.

(a) Multi-Layer Perceptron Parameters



(b) XGBoost Parameters



(c) Support Vector Machines Parameters

Figure 1: Hyperparameters for each algorithm

Model Selection

For each algorithm, two of the highest performing sets of model parameters were selected. In each case the top performing set of parameters and a member of the next four highest performing sets of parameters with the lowest model complexity was selected. This selection was done manually based on the results of the Bayesian Optimization. The chosen parameters can be viewed in the tables below.

Model Name	Max Depth	N Estimators
XGB Top	6	50
XGB Simple	4	59

Table 1: Parameters for XGBoost

Model Name	C	Gamma
SVM Top	0.00022594357702209782	0.04920395356814509
SVM Simple	0.00001	1

Table 2: Parameters for Support Vector Machines

Model Name	Alpha	Layer One Size	Layer Two Size	Layer Three Size	Layer Four Size
MLP Top	0.03375	11	142	33	99
MLP Simple	0.04579	27	196	49	126

Table 3: Parameters for Multi-Layer Perceptron

Testing

Models were assessed on precision, accuracy, and the amount of money (theoretically) made or lost after using the model to short the S&P 500 for 3.5 month period. The "best-performing" algorithms are the ones that makes the most profit, while maintaining reasonably high accuracy and precision. The Accuracy and Precision scores were generated from `ci_generator.py` with the ±s being 95% confidence intervals for the mean score. These scores were genereated on the same dataset optimization was performed on. The "Money

Gained or Lost" column was generated using training data from January 1st, 2015 to December 31st, 2019 and test data from January 1st, 2020 to April 17th, 2020 in `model_assessment.py`.

Model Name	Accuracy	Precision	Mean Fit Time	Money Gained or Lost
XGB Top	0.49717 (± 0.0267)	0.44731(± 0.0287)	80.502 (± 38.590)	+$1124.40
XGB Simple	0.46698 (± 0.0257)	0.40733 (± 0.0285)	80.211 (± 38.645)	+$4577.80
SVM Top	0.52075 (± 0.0481)	0.18491 (± 0.1991)	80.864 (± 39.130)	+$0.00
SVM Simple	0.51351 (± 0.0481)	0.18491 (± 0.1991)	81.094 (± 39.365)	+$0.00
MLP Top	0.49906 (± 0.0250)	0.44884 (± 0.01814)	81.278 (± 39.327)	+$616.81
MLP Simple	0.46792 (± 0.0203)	0.41392(± 0.0278)	82.536 (± 39.663)	+$1641.80

Table 4: Assessment Metrics for each Model

Conclusion

Based on the assessment metrics, XGBoost is the clear winner for performance on this dataset with the highest (actual) precision and accuracy scores. The clear loser was Support Vector Machines. Although SVM looks like it was the best performing in terms of accuracy, it only gained this lead by becoming a Stock Market Bull and predicting that the market would go up every day (i.e. predicting all `False`). The MLP Classifier performed reasonably well, but was outperformed by the XGBoost model in most cases. In terms of fit time, the models are about equal, with the majority of the fit time actually coming from the text processing and vectorization. Because the application of these models would be in algorithmic trading, the metric that matters the most is how much money each algorithm could make if deployed in such a setting. Therefore XGBoost is the clear winner here, making the most profit on average. The true best model however, is a combination of all of them. By using a trading strategy that takes into account the predictions from all of the models and adjusts position size accordingly $17,396.64 can be made.

Acknowledgements

Thanks to Chase Clough for reviewing this paper.

References

[1] *Barron's News*. URL: `https://www.barrons.com`.

[2] Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[3] *Bloomberg News*. URL: `https://www.bloomberg.com`.

[4] Fernando Nogueira. *Bayesian Optimization: Open source constrained global optimization tool for Python*. 2014. URL: `https://github.com/fmfn/BayesianOptimization`.

[5] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[6] *RBF SVM parameters*. URL: `https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html`.

[7] *TextBlob*. URL: `https://textblob.readthedocs.io/en/dev/index.html`.

[8] *The Wall Street Journal*. URL: `https://www.wsj.com`.

[9] *The Wayback Machine*. URL: `https://web.archive.org/`.

[10] *Varying Regularization in Multi-Layer Perceptrons*. URL: https://scikit-learn.org/stable/auto_examples/neural_networks/plot_mlp_alpha.html.

[11] *XGBoost*. URL: https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.sklearn.