MF850 Final Project

Predicting Stock Returns with Machine Learning

Issy Anand, Allen Chacko, Robbie Cook, Henry Gagne

Team : Snorlax

Abstract

This project aims to predict stock trends and stock returns using machine learning techniques. First, we prepare the data for input into the machine learning algorithms. This entails removing features with no or low importance, adding new features, standardizing the data, and imputing missing values. After the data is cleaned, we need to choose two machine learning algorithms: one to solve the classification problem of predicting stock trends, and one to solve the regression problem of predicting stock returns. After testing a number of different algorithms on our data, we determine that, based on a combination of R-squared values and computational performance, our choice for both problems is the Random Forest algorithm. Next, we tune the Random Forest hyperparameters in order to minimize the error as much as possible. Then we train the model, using a recursive rolling window technique to split the time series data into training sets and out-of-sample test sets. This must be done in order to preserve the correlation of the observations through time. Finally, we are able to use our trained models to make predictions on new data.

1. Introduction

Predicting stock returns is one of the oldest problems in finance. Everyone would like to be able to successfully predict whether a stock price will increase or decrease in order to maximize his or her wealth. In recent years, as computational power has increased and as more financial data has become available, machine learning methods have been applied to this problem with some success. To this end, we will attempt to do the same in the following project. We would like to solve two problems. The first is to use regression techniques in order to predict stock returns, and the second is to use classification techniques in order to predict whether a stock price will grow or fall.

We employed several steps in order to solve our given problems. The first involved cleaning our data, due to missing in some cases a large percentage of the inputs. Next we attempted to identify the important columns from our dataset and cut it down to avoid overfitting. Following this we ran through several models for both the regression and classification problems before settling on the ones which gave the strongest performance.

2. Data Preparation

2.1. Custom Features

We decided to define new compound features to better capture company fundamentals and technicals. The P/E ratio can be a good indicator of the relative value of a stock, so we created additional columns to capture those. The dataset included four types of earnings, so a P/E ratio was calculated for each. Because these columns would later be filtered for high collinearity, we did not need to be too concerned about introducing it with overlapping definitions. For each new feature, we removed the existing features it was calculated from, due to the same issue of collinearity, as any new column will be highly correlated with its constituent parts. The last simple ratio we included was P/B, or price to book. This ratio is closely linked to a company's overall financial health, so it could be useful in training the models. Last, we captured the 1 Month momentum, because on a short term basis, momentum is a key driver of stock prices.

Custom Feature	Description
PtoE_f12	Earnings Per Share (Diluted) - Excluding Extraordinary Items - 12 Months Moving Avg.
PtoE_fxq	Earnings Per Share (Diluted) - Excluding Extraordinary items
PtoE_pxq	Earnings Per Share (Basic) - Excluding Extraordinary Items
PtoE_x12	Earnings Per Share (Basic) - Excluding Extraordinary Items - 12 Months Moving Avg.
PtoB	Price to Book Ratio
mom1m	1 Month Price Momentum, in Log Return

Table 1: Custom Defined Features

2.2. Scaling Data

After defining custom features, we used the Scikit-learn standard scaler to rescale the features by removing the mean and scaling to unit variance. This helped standardize the feature data, because many different orders of magnitude were present in the data, so by first scaling, we ensure that we do not only capture behaviors from the largest magnitude columns. It is crucial to do this step first in data preprocessing, before applying PCA for dimensionality reduction.

2.3. Selecting Features for Removal

We used several criteria in a first pass to determine features for removal. The feature selection code came from a Feautre_Selector package implemented by Will Koehrsen, which we sought out due to its implementation of a Gradient Boosting Machine. The GBM library quickly trains a decision tree model

1) For columns with a large percentage of missing data, we decided to drop these from the features as opposed to imputing them. By setting a 20% missing data cutoff, we eliminated 8 columns.
2) Highly Correlated Features: Many features in the model were highly collinear, so we removed any features with a correlation magnitude greater than .95. This eliminated another 19 columns
3) Zero/Low Importance Features: We identified zero and low importance features in the regression and classification problems by using a gradient boosting machine from the LightGBM library. The evaluation metric used for the classification was ‘auc’ and ‘mse’ for the regression. Due to the randomness from the gradient training method, the features identified for removal can change slightly each time, but overall this does not impact the model accuracy. This method identified 9 additional features for removal

Justification	Features Removed
Missing >20% data	'cshopq', 'tfvaq', 'tfvceq', 'tfvlq', 'xaccq', 'xrdq', 'intpny', 'txpdy'
Highly Correlated	'm_low', 'm_high_adj', 'm_low_adj', 'SP500WeeklyLow', 'SP500WeeklyClose', 'cshfd12', 'cshfdq', 'cshprq', 'ibcomq', 'ibmiiq', 'ibq', 'lltq', 'ltq', 'niq', 'xoprq', 'gind', 'gsector', 'gsubind', 'PtoE_pxq'
Zero/Low importance	'cshprq', 'fyearq', 'ibcomq', 'tfvceq', 'ibq', 'txdbclq', 'esopctq', 'rdipq', 'gsector'

Table 2: Removed Features

2.4. Imputing Missing Values

After removing selected features from the dataset, we used the Scikit-learn SimpleImputer mean method to fill in the missing values on the remainder. Due to the volume of the data, we found the run-time of iterative imputers to be too long, and these also seemed to be in experimental phases. So for this step we stuck with the more well established method.

2.5. Dimensionality Reduction

In order to further reduce the dataset and increase the efficiency of training the random forest models, we applied PCA to our dataset after imputation. We set a high threshold for the cumulative variance to retain in the PCA fit by selecting a fixed number of components (53) that explained 99.9% of the variance. This was sufficient in reducing the dimensionality of the training data, while still retaining the majority of its important characteristics. **Note**: We had to remove this section from our code due to runtime errors on the unseen test data file. We will leave the section in the paper to show that the work was done despite not being implemented.

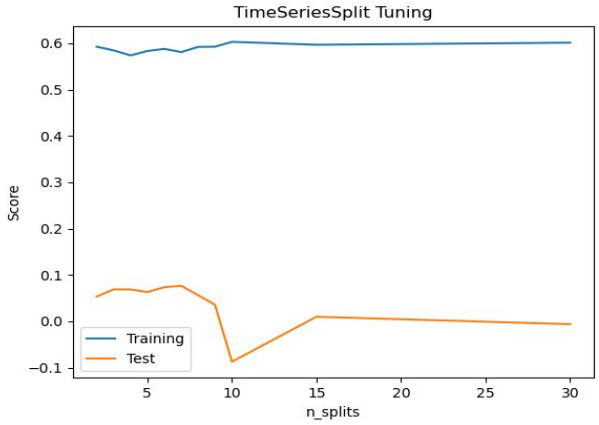
3. Methodology and Justifications

3.1 Data Splitting

In a typical machine learning problem, the dataset is split into three sets: one each for training the model, validating the hyperparameters, and doing a predictive test on out-of-sample data. There are many methods for choosing exactly how to perform this split operation, and there are other variations in which more than three sets may be used.

However, time series data poses a time correlation problem that is not present in other datasets. When splitting the data, we must take into account the fact that newer observations are not independent of older ones. This means that splitting the data in the usual way will not capture this component of correlation over time.

Scikit-Learn addresses this problem with TimeSeriesSplit, which works as follows: the model is trained on the earliest observations and tested on observations that occur later in time. This process is repeated with a training set that grows in a rolling fashion, with the out-of-sample test set remaining a constant size but getting pushed further into the future as the training set grows. Gu et.al.(2020) use a similar procedure for splitting their time series data, although they do not specifically mention if they are using TimeSeriesSplit. We can pre-specify the number of times that we would like to split the data using the n_splits parameter, and like all other parameters in our model, this can be tuned. We found that the best training and test scores resulted from using n_splits = 7.



Graph 1: TimeSeriesSplit Tuning

3.2. Choosing a Model

There are many different supervised learning techniques available for the simple classification and regression problems. We tested our training data using these various techniques and concluded that random forest regressors and classifiers tend to outperform every other learning algorithm for our dataset. A detailed analysis of this is presented in the following section.

Random forest is a supervised learning algorithm that builds multiple decision trees and merges them together for an accurate and robust prediction. The term “forest” is an ensemble of decision trees, trained using the bagging method. The term “random” is due to the fact that the algorithm searches for the best feature among a random subset of features while splitting a node instead of selecting the most important feature.

Random forest does not rely on any prior assumptions regarding underlying distribution. Since random forests are non parametric it can handle skewed and fat tailed distributions, as well as categorical data that are ordinal or non ordinal.

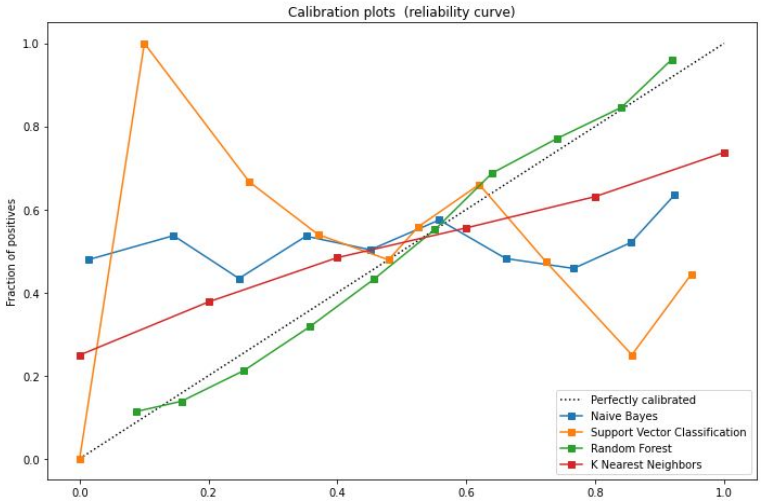
Although sklearn has pre implemented functions for random forest algorithms, the availability of hyperparameters helps us to fine tune our model. The hyperparameters can be used to increase our model accuracy or reduce the computational turnaround time.

We fine tune our model using the following hyperparameters of random forest algorithms which are implemented by Sklearn.

1. n_estimators: This hyperparameter is the number of trees the algorithm builds before deciding on the averages of predictions.
2. max_features: The maximum number of features a random forest algorithm considers while spliting a node.
3. min_sample_leaf: Determines the minimum number of leafs required to split an internal node.
4. n_jobs: This tells the computer how many processors it is allowed to use, a value of 1 means use one processor and a value of -1 means that there is no limit.

3.3. Classification Model

For the classification problem, we needed to add a column to the dataset that would contain the classification associated with each row. To create the classifier, we took the log return from the next month, and set the class to 0 if the return was negative, and 1 if it was positive. The data was then preprocessed in the same manner as the regression, and fed into a random forest classifier. We chose random forest for the classification for a variety of reasons, but mainly because it showed the most consistent results for an efficient runtime. In testing multiple classifiers on the same set of training data, we first look for a model that will classify the correct proportion of each class. Even if the accuracy isn't perfect, a well calibrated model will classify the same proportion of 1's to 0's in the test data as it was given in the training data. After running the calibration plot for multiple classifiers, including Naive Bayes, Support Vector Classification, Random Forest, and K Nearest Neighbors, we see that both Naive Bayes and SVC give choppy results, where we would expect a relatively straight, upward sloped line. Both Random Forest and KNN show promising calibration results, but Random Forest was selected by our group due to its more efficient runtime with comparable accuracy to KNN.



Graph 2: Calibrating Classification Models

To further reinforce our model selection, we calculate the precision and recall scores, for the four models shown above. We see that with the exception of Naive Bayes' recall score, Random Forest performs the best out of all models.

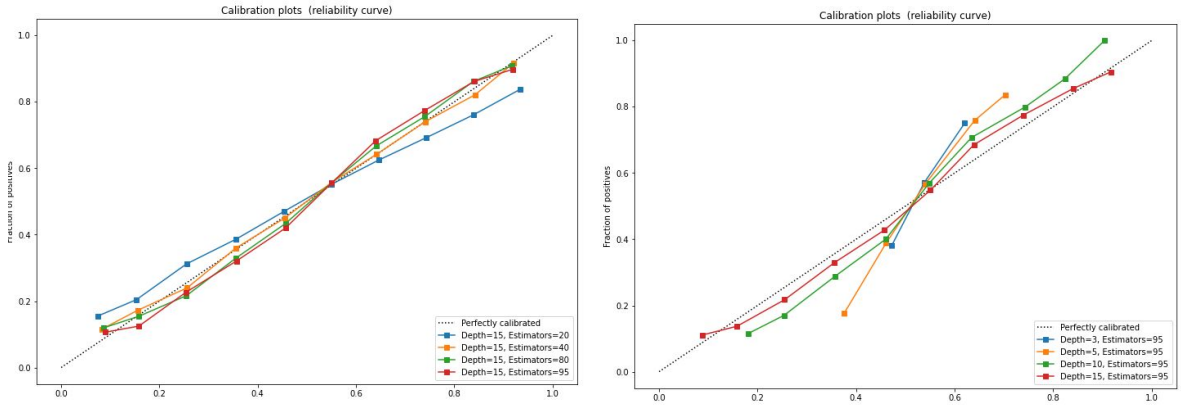
Model	Accuracy Score	Precision Score	Recall Score
Support Vector Classification	54.65%	55.38%	68.60%
Naive Bayes	52.39%	52.44%	96.64%
Random Forest	63.51%	62.61%	75.11%

K Nearest Neighbors	59.84%	61.09%	63.97%

Table 3: Classifier Accuracy Scores

3.4. Hyperparameter Tuning - Classification

We tuned the hyperparameters of the random forest using SKlearn's GridSearchCV. This function returns the parameters that maximizes the score of the test data. By running it for tree depth from 1 to 20, and the number of estimators ranging from 10 to 100, we found an optimal parameter set of max_depth = 15 and n_estimators = 95. To further reinforce that this choice of hyperparameters is optimal, we can again use the calibration plots, and see how the fits change as we vary max_depth and n_estimators. Starting with n_estimators, we see that for a max_depth=15, the n_estimators does not substantially change the calibration result. The precision and recall scores increase slightly with each addition to the estimators, so we went with the highest one. For the max_depth parameter, a lower depth speeds up the algorithm considerably, but does so at the cost of accuracy. By setting a max_depth of 15, we have a well calibrated model that still runs decently efficiently.



Graph 3: Calibration of Random Forest Hyperparameters

3.5. Regression Model

For the regression problem, we needed to create a new column that we could fit as our target output, as the m_ret column contains the monthly adjusted returns for the previous month. We therefore took the returns from the adjusted close for each company and assigned this to a new "target_ret" column, which we used as our Y variable in the regression.

We decided to use Random Forest regression after trying several models and deciding it was the most appropriate. Decision Tree, Gradient Boosting regression, and K-Nearest Neighbours all gave worse results, so we decided to proceed with Random Forest.

Model	Training Score	Test Score
Random Forest	58.11%	7.64%
Gradient Boosting	40.73%	6.63%
Decision Tree	30.95%	3.93%
KNN	25.33%	-9.71%

Table 4: Regression Accuracy Scores

3.6. Hyperparameter Tuning - Regression

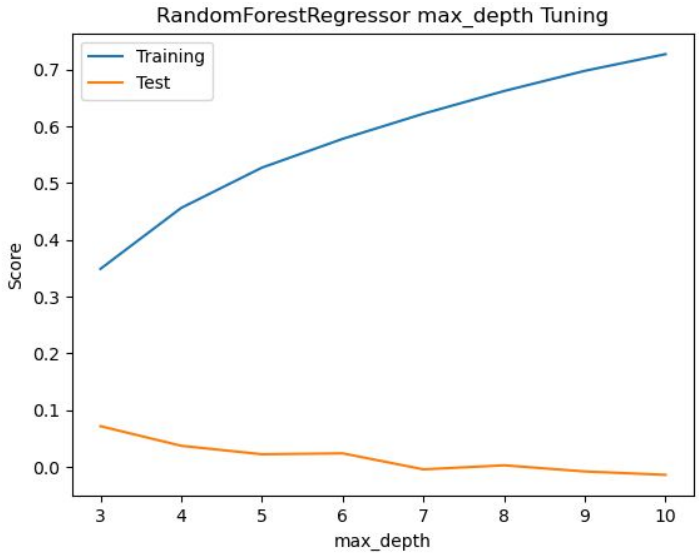
For performance reasons, we tuned the hyperparameters of the random forest regressor using RandomizedSearchCV as opposed to GridSearchCV. Instead of doing an exhaustive search through the parameter grid the way GridSearchCV does, RandomizedSearchCV only tries a pre-specified number of samples. There is a small trade-off in accuracy, but the performance gains are large. After several iterations of RandomizedSearchCV as well as some manual trial and error, the parameters that resulted in the best balance of training score, test score, and computational effort are as follows: max_depth=15, n_estimators = 150, min_samples_leaf = 2, and max_features = 'sqrt'.

4. Results and Analysis

4.1. Regression

As established from the test scores generated by the Random Forest regressor, we did not get the strongest results. These were generally close to 0 with some splits giving negative results. Whilst this was not very encouraging, we did try several adjustments in hope of improving the accuracy such as increasing the number of splits, varying the Random Forest parameters, excluding more/less columns, however none of these led to significant improvement. We therefore concluded our poor results were down to the complexity of the problem, it is of course notoriously difficult to predict stock returns.

For one particular example of the regressor (random_state = 55), using the hyperparameters described in an earlier section of the paper, we saw a training score of 56.01% but only an out-of-sample test score of 7.64%. However when looking at the scores in Gu et.al. (2020), their monthly Random Forest regressor had an out-of-sample score of 0.33%. Our results are actually quite good in comparison. For various values of the hyperparameters, there was always a trade-off between training score and test score. Higher training scores led to very low or even negative test scores, which we felt was undesirable and probably indicative of an overfitted model. The max_depth parameter is a perfect example of this behavior as seen in the figure below:



Graph 4: Hyperparameter Tuning - max_depth

4.2. Classification

We have a few metrics we can use to assess the performance of the classifier. First, we looked at the accuracy, precision, and recall scores for Random Forest, using the optimal hyperparameters from GridSearchCV. The classes were relatively balanced, with 52% of rows classified as 1, and 48% classified with 0, so the worst case baseline for our accuracy is 52%. Assuming the training and test data have roughly the same proportions of classes, a trivial classification of assigning each observation a 1 will coincidentally classify 52% correctly. Any improvement from this baseline is a good indication that our classification is outperforming random chance. Below is the output for a sample run of the time series split model training. Immediately we notice a very high training score for the earlier splits, which is consistent with expectations. We are using a relatively high max_depth for our random forest, due to the high dimensionality of the dataset, so when the data used for training is small, the model can easily overfit. Although typically these high training scores could be worrisome, we have already calibrated the random forest using a reliability curve, so we have confidence in our chosen hyperparameters.

For the test scores, we see a consistent performance of roughly 62% accuracy, for all time series splits. So our model does outperform the baseline. The test scores are relatively robust, and are not overly sensitive to the training data used, as long as a sufficiently large sample is given for training.

Split	Training Score	Test Score
1	97.24%	61.58%
2	93.95%	62.50%
3	91.77%	62.78%

4	89.70%	62.77%
5	88.19%	62.11%

Table 5: Training and Test Scores for Random Forest Classification

5. Improvements and Further Research

One of the main limitations of our random forest model is overfitting, this was more predominant while running the random forest regressor than the classifier. A large number of trees can make the random forest algorithm too slow and ineffective for real time predictions. Another major drawback of random forest is that it is not a descriptive tool, meaning it lacks explainability compared to other econometric models.

The methods discussed above use supervised learning techniques to predict the stock return and to judge whether the stock price will grow or fall in the next time step. These techniques are not adequate when it comes to long-term goals and delayed reward. There is a growing interest among practitioners to apply the concepts of reinforcement learning and agent based modelling to better predict the market. Neural networks and deep learning techniques have also shown superior results when making price predictions, as seen in Gu et.al. (2020).

Further improvements can be made by taking advantage of the built-in feature importance functionality of RandomForestRegressor and RandomForestClassifier in SciKit-Learn. After training the predictor, we are able to see which features had the biggest contribution to the training score. This means that we can remove the features with the smallest contribution and re-train the model, hopefully seeing improvements. This is another way of reducing the dimensionality of our problem as we did in this case with PCA. A future version of this project should take this into account.

6. Conclusion

In conclusion, we were able to achieve our goal of training machine learning models and using them to make predictions in regards to both a classification problem and a regression problem. Along the way, we learned how to choose the most suitable algorithms among the many that are available. We also discovered that the data itself requires cleaning and engineering before it can be fed into the algorithms. Finally, it became clear that there is no “right answer” to solving these kinds of problems. There are a lot of different ways to arrive at similar solutions, and there are potentially a few solutions that are better than others.

References

Shihao Gu, Bryan Kelly, and Dacheng Xiu, “Empirical Asset Pricing via Machine Learning”, *The Review of Financial Studies*, Volume 33, Issue 5, May 2020, Pages 2223-2273.

Will Koehrsen, A Feature Selection Tool for Machine Learning in Python, *towordsdatascience.com*