Teaching a Machine to Trade Stocks like Warren Buffett (Part 2)

Utilizing machine learning for stock fundamental analysis

Marco Santos
Jan 17, 2020 · 11 min read ★

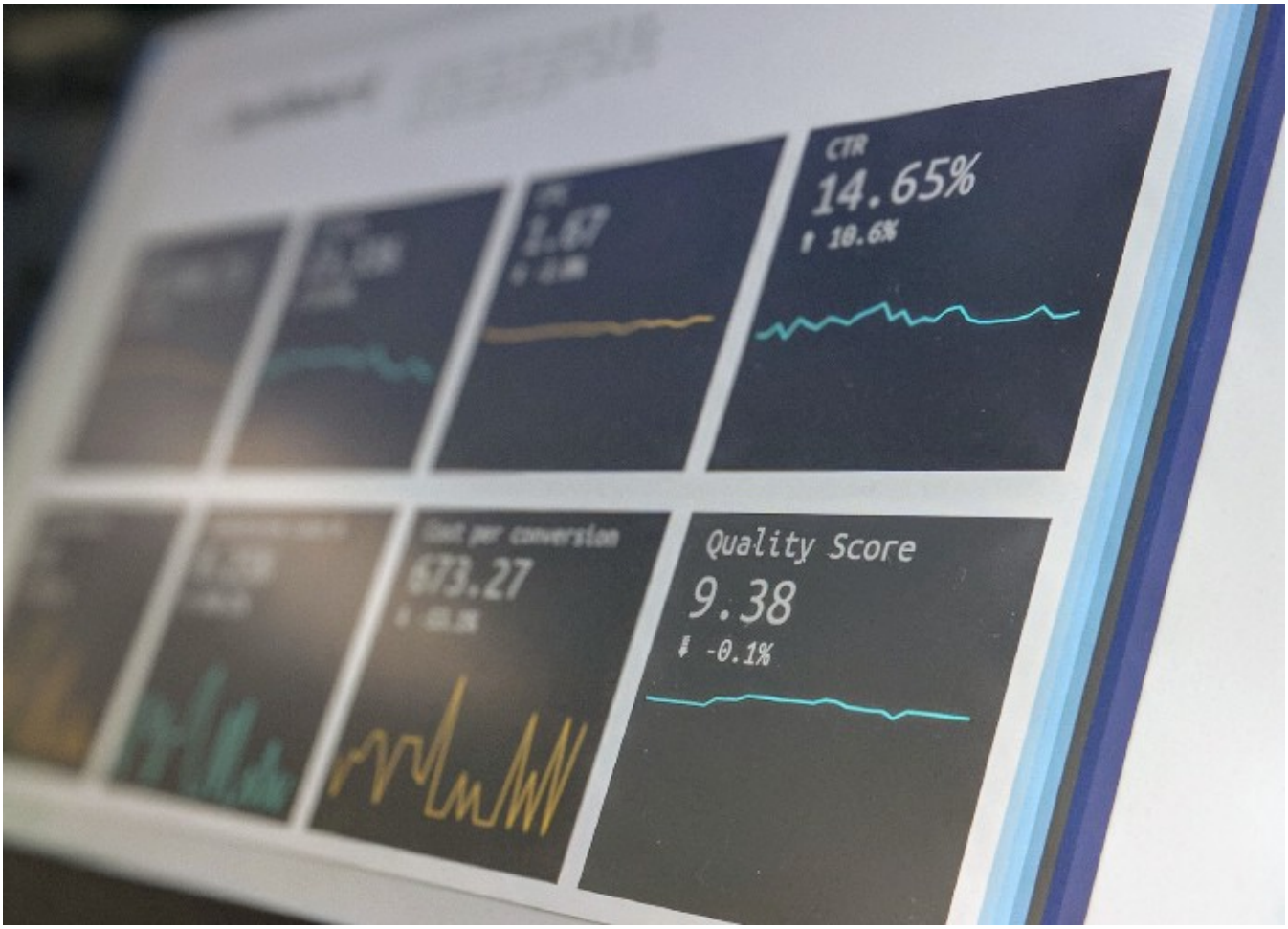

Photo by Stephen Dawson on Unsplash

Editor's note: This article is intended for informational and educational purposes only and does not contain financial advice.

Click below for Part 1:

Teaching a Machine to Trade Stocks like Warren Buffett, Part I

Utilizing Machine Learning for Stock Fundamental Analysis

medium.com

In the previous part, we set out our a goal: to create a machine learning classification model that utilizes fundamental analysis on stocks by analyzing Quarterly Report data, thereby determining whether a stock is worth investing in at that time. In order to do this, the necessary information went through a process of data engineering involving data collection, cleaning, formatting, visualization, and feature selection. This entire procedure prepared our data for the next exciting phase: data modeling!

Machine learning modeling involves taking in data to train models in order to make predictions based on that data. However, these models cannot take in just any data we give it — it has to meet a specific standard. This standard was subjectively set once we had decided what kind of problem we were solving. Every problem requires their own set of unique data and most of the time this data has to go through the entire data engineering process.

Our Data

Since we were explicitly examining quarterly reports, we decided to evaluate a stock's investment value by observing their QR performance in relation to their previous QR and storing it as a percentage change. The class labels of `Buy`, `Hold`, `Sell` were determined by observing the next QR's `Price high` and `Price low`, and whether or not they increased or decreased by a significant amount.

The two sets of data we have: `percentage change values` and `class labels`, will constitute X and Y values respectively. The percentage change values (X-value) is

known as the `independent variable`. The class labels (Y-value) is known as the `dependent variable`. These variables will be scaled in order to improve the performance of our models. Then, the scaled data will be split into training and testing sets to evaluate our models. Once the data is ready, we will be able to finally fit the model to our data then make some predictions/classifications!

In the last part, we created two different sets of data with slightly different top ten feature sets in each (the correlated features vs. the tree classifier's feature importances). This gave us the opportunity to compare and contrast the model performances based on the data given to the model. One dataset may actually perform significantly better than the other. This may sound like extra work, but in actuality, we just have to change the data loaded into the model, which requires altering just a single line of code.

Machine Learning Classifiers

For the purposes of finding the best performing Machine Learning classification model, we will be running our data through multiple models. Each model has its own advantages and disadvantages. To ensure we're using the optimal model for our data, we have to try out each classifier. Don't worry, once we write the code for one model, we can just copy the same code for each new model with just a few one or two-line modifications.

The classification models we will be using:

- AdaBoost
- Decision Tree
- Dummy (Baseline model)
- Gradient Boost
- K Nearest Neighbors (KNN)
- Logistic Regression
- Naive Bayes

- Random Forest
- Support Vector Machine (SVM)
- Extreme Gradient Boosting (XGBoost)

That's a lot of models! Each of them will be trained and fitted to our dataset in order to determine the best performing model.

Dummy classifier

To measure the performances of each model, they will need to be compared to a baseline model. This is where our dummy classifier comes in. The dummy classifier functions similarly to randomly guessing a class label for each QR. It's usually based on probability — class labels would be determined based on their sample size.

With the dummy classifier established as our baseline model, a performance standard has been set for every model. Each model has to be able to outperform the dummy classifier in order to be considered a viable model. If they are not able to outperform the baseline model, then the model is no better than random guessing.

Coding the Classification Models

Now that we have our data ready and have established the models we will be using, we can code our first classifier. This will be our baseline model (dummy classifier) in order to set the standard to beat.

Importing libraries and loading the data

Here we load in Python libraries we need to develop a dummy classification model. Most of these libraries will remain the same for the rest of the models — just replace the `DummyClassifier` import line. When it comes to importing our data, we have the option of choosing which dataset we want. This will be important later if we wish to compare the model performances of the two different datasets.

Scaling and train, test, splitting the data

We scale the data here to improve the overall performance of most of our models. The method of scaling to use is up to you. Each scaler has their own advantages (click here to see the comparisons of each scaler).

The data is then split into train and test sets for the X and Y variables, which we will need to fit and evaluate our models.

Creating the dummy classifier

We create the Dummy classifier by instantiating the classifier, then fitting it to our training sets. Then we use the test set to make predicted labels to compare to the actual labels. Finally, we print out a classification report which contains the evaluation metrics we need:

The evaluation metrics from the classification report function for our Dummy Classifier

Evaluation Metrics

Since we're dealing with an imbalanced dataset, we cannot rely on accuracy as an evaluation metric with our models. This why it is not included in the classification report. Accuracy may be an important measurement but due to the nature of our problem and the class imbalance, it will be ignored in favor of *precision* and *recall*. Click here for more information.

Precision > Recall

Why are we choosing Precision over Recall? Precision deals with the number of *false positives.* When it comes to investment decisions, we don't want the classifier to

incorrectly classify a `Sell` as a `Buy`. On our conservative investing strategy, we would rather miss an opportunity to invest (`Recall`) than invest in the wrong stock (`Precision`). Obviously there is a balance to having each, which is where the `F1-Score` comes into play. However, the overall judgment for each classification model will be based on *increasing its* `Precision` *score* relative to the baseline model.

Confusion Matrix

It may be better to visualize the results of our model. For each classifier, we will be visualizing their confusion matrix for better interpretability. The confusion matrix will show us the amount of QRs that were misclassified.

Running this will gives us:

The Confusion Matrix for our Dummy Classifier

With the confusion matrix visualized, and our baseline model created and evaluated, we've officially finished our first classification model!

Constructing More Classification Models

Building more models may seem like a daunting task but don't worry — as we said before, to create more models all we really have to do is copy our baseline model's code and then modify a couple of lines. Here's the code for one model (which is similar to the baseline model):

The entire code for building a model (along with evaluation metrics and confusion matrix)

As we can see, there are really only two or three lines in the code that need to be modified to build each model, all thanks to Scikit Learn's accommodating library.

The two or three lines that we would need to change are:

- `with open("top10_df.pkl", "rb") as fp:` — only modify if testing out the other dataset of different features.
- `from sklearn.ensemble import AdaBoostClassifier` — change these imports to the appropriate library and its respective classifier.
- `clf = AdaBoostClassifier()` — change this variable to the appropriate classifier.

Everything else should remain the same and should run with no issues. (*Check the Github at the end of this article for an overview of each model*).

Grid Searching — Parameter Tuning

When it comes to adapting the models to the datasets, we can leave the models as is or we can enhance their performances by using `Grid Search`. By using this method, we will be able to appropriately adapt the parameters of each classification model to our dataset.

Out of all the things we had to do so far to develop our classification models, grid searching is the most time-consuming. Not because of the amount of code we have to write (not much), but because of the number of iterations for each parameter combination. Some models take longer to grid search than others but the overall process will still take a considerable amount of time because we will be grid searching every model we have.

The code for grid searching is essentially identical for each model and we would only really need to change a couple of lines of code.

The code to run Grid Search for each classification model

As you can see, there's not much code that needs to be written and in fact some code here we used before (the classification report and confusion matrix).

What does need to be changed:

- The `params` variable, which will be a dictionary with the dictionary's keys corresponding to the respective classifier's parameters and the dictionary's values being the different values we want to try out for each iteration.
- `clf` should have been altered long before when the variable was initially changed to accommodate each new classifier.

GridSearchCV's Parameters

In regards to `GridSearchCV`'s own parameters, those need to be set to suit our own personal requirements. The `clf` and `params` were handled above as we can see. But for the other parameters:

- `return_train_score` is set as `True` in order to observe the results for the evaluations later on.
- `verbose` is set to `5` so we can see the progress of each iteration and their scores.
- `scoring` is set to `'f1_macro'` because we are prioritizing the F1 Score when determining the best parameters.

F1 Score Macro Average

The reason for the `F1 Score` being prioritized over `Precision` is simply due to the quantity classified. When `Precision` is prioritized, the `Precision` score may increase significantly but the amount of QRs being classified as certain classes are greatly reduced. In our case, the classifier would rarely make a classification of `Buy` or `Sell`, but many in the `Hold` class. This is because the `Hold` class was the majority class. We don't want our classifier to rarely tell us when to invest, we would want at least a decent amount of opportunities which would fall under the `Recall` score.

To strike a balance between `Precision` and `Recall`, we would prioritize the `F1 Score`. The reason we chose the `Macro Average` over the `Micro Average` is because the Macro disregards the class imbalance. The `Hold` class, being the dominant class, tells us close to nothing when it comes to investment activities — we would prefer to know whether a stock is a `Buy` or `Sell`. Overall, using the `F1 Score Macro Average` will allow `GridSearchCV` to find the best parameters under our preferred conditions.

Running and Evaluating Every Classification Model

Now that we have the data, the classifier's code, evaluation reports, and the code for grid searching, we can apply them to each classification model. This may take a while but we will know for sure at the end of this which classifier performs the best when classifying a stock. There's also the option to use `Sklearn's Pipeline` library to streamline the process but we should go through each model to really comprehend each step. (Check the Github at the end to see how we ran each model).

Saving a classification model

Once we've decided which model was the best for our data, we can save the fitted model with the correctly tuned parameters. Look at the following code to see how to save a model (just two lines of code):

```
# Importing the library
from joblib import dump

# Saving the classification model as a file named 'xgboost.joblib'
dump(clf, "xgboost.joblib")
```

This code will essentially export our model for use in other files.

The best performing model

In our case, the best performing model came from the XGBoost Classifier using the Top 10 important features from the tree classification dataset. This classifier's classification report is as shown:

XGBoost's classification report

As we can see here, the XGBoost model had an increase of 10% for the *Buy* class and 20% for the *Sell* class when compared to the baseline model (seen before). This is a significant improvement over the baseline model. The reason for the numbers not being closer to 100 is because of the nature of our question. In many ways, we're trying to predict the future. If our classifier is able to determine if a stock is worth investing in with 47% accuracy (or Precision score), then we should consider that a significant achievement!

Classifying New Data

Let's say we wanted to use our classifier to make predictions on a new QR. How would we accomplish that? The best option would be to find the values of the latest QR and the previous QR, then observe percent changes between them so that they match the format from our own dataset.

The new data can be acquired from numerous websites that report QRs. We could either web-scrape those values or enter them manually. Stockpup.com may also have the latest QRs and, for this project, we'll be using the latest values from that site. In the previous part, we removed the latest QRs from each stock because there was no future

Price high and low to use for our class labels. So these QRs remain unlabeled but now we might be able to label them using our classifier's prediction.

Preparing our Data and model

The data we are using is again from Stockpup.com but now we're specifically seeking out the latest QRs. Ideally, we would be gathering the most up-to-date QR from other web sources via web-scraping or entering them manually. This is because stockpup.com does not always have the latest QR available.

Helper functions for our latest QR and visualizations

When working with the new data, in order to cooperate with our classifier, we must scale it because that's how we initially trained the classifier. The data must be formatted to include the percent changes, the correct feature columns, and scaled values in relation to the rest of the data. Once we have all that, we can classify the latest QR.

Classifying the latest QR

For this example, we will be using the latest QR from AMD:

Running this code will also create a pie chart with the prediction probabilities

Prediction Probabilities for AMD

Based on our classifier, AMD has a 42.5% chance of being a `Hold`. The most dominant probability will determine how the classifier determines a stock's designation.

Conclusion

Deciding a stock's worth based on Quarterly Reports is not a new achievement. For most Fundamental Analysts, the strategy we utilized may be considered too simple or outdated. But for the purposes of learning machine learning classification, it was enough. We could potentially add more features to improve the model or alter the features we have based on new strategies.

There are also techniques to streamline the process such as creating a `Pipeline`. Or we could even experiment with neural networks to see if they could perform better than our current models.

Those options will be for another article in the future. So be on the lookout for more!

In the meantime, we hope you learned more about the creation and evaluation of machine learning models. Feel free to alter the code to experiment with your own strategies. There's a multitude of possibilities to implement and we now have machine learning to help us backtest any strategy we can think of!

*Follow me on Twitter: @_Marco_Santos_*

Resources:

marcosan93/Stock-Performance-Predictor-2.0 github.com	
Teaching a Machine to Trade Stocks like Warren Buffett, Part I Utilizing Machine Learning for Stock Fundamental Analysis medium.com	