# Insider Trading Data Analysis and Optimal Portfolio Creation

*MATH4570*

Brian Reicher Maxwell Arnold Ryan McDonald Kyle Rudman

December 11, 2021

## Abstract

*In theory, analysis of insider trades presents the opportunity to monetarily capitalize on acquistions and dispositions that are quite well informed. With the hope to create a financial trading algorithm which confidently brings about high profits, we implemented machine learning algorithms to ensure our predictions' success. Here, we present an in-depth approach into using simple and advanced classification measures on a data set of 400,000+ insider-trades from 2004-2021 in order to optimize our goal of profiting from insider trades.*

## 1 Introduction

For our research, we hoped to be able to predict forward returns on insider trading exchanges, and in doing so, find an optimal algorithm to predict the returns on our own trades. Insider trades are stock exchanges made by those who work at companies or in government, which must be disclosed to the public according to national law. Resulting from a select handful of machine learning techniques, we hoped to select the most successful of the cluster and utilize it to extrapolate the average returns we could expect. We decided to implement our analysis specifically on insider trades because we believed that these transactions would hold better returns than the "average" consumer trades, given that business/government workers might be more inclined to make educated decisions. Historically, insider trades have led to massive returns on investment, and we were hoping to be able to harness these returns ourselves without having the company information which these executives have access to.

## 2 Formulation

To enact our goal in obtaining an optimal investment strategy, we analyzed a number of algorithms and techniques to draw the most concrete conclusions. We knew we wanted to be able to confidently assume whether a trade would return positively or negatively on the initial investment, so a classification technique would clearly work best over a generalized linear/non-linear regression predictor or some sort of gradient descent algorithm. Here, we have broken down the selected algorithms:

### 2.1 Logistic Regression

Logistic regression, perhaps the simplest form of binary classification, implements a training set of data and specified number of features to create a decision boundary. We can fit the listed data to a sigmoid function, defined:

$$h_{\vec{\theta}(x)} = S(\vec{\theta}^T \vec{x}) = \frac{1}{1 + e^{-\vec{\theta}^T \vec{x}}} \tag{1}$$

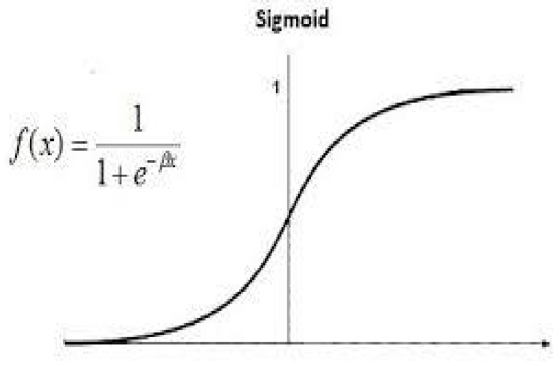

Figure 1: Graphical representation of the sigmoid function

This sigmoid function can then be used to map any real value into a probability $p_i \in [0, 1]$. Using the binary function:

$$y(\vec{x}) \begin{cases} 1 & h(\vec{x}) \geq 0.5 \\ 0 & h(\vec{x}) < 0.5 \end{cases} \tag{2}$$

where 1 and 0 represent different sides of a binary option. Essentially, we want to maximize the likeyhood function $L(\vec{\theta})$ in order to find $\vec{\theta}$, which can be used to find this decision boundary. Formally stated:

$$\vec{\theta}^{optimal} := \text{argmax}(L(\vec{\theta}) = \{\vec{\theta} | L(\vec{\theta}) \textit{ is the minimum}\} \tag{3}$$

This can be extrapolated to the logarithmic likelyhood function, $\lambda(\vec{\theta}) = \ln\Big(L(\vec{\theta})\Big)$. Further, we can define the log-cost function, or cross-entropy function, which is essential the structure of the logistic regression problem. In finding the best fit of the model, we minimize the following:

$$J(\vec{\theta}) := -\frac{1}{n}\sum_{i=1}^{n}(y^{(i)}\ln\Big(h(\vec{x}^{(i)})\Big)+(1-y^{(i)}\ln\Big(1-h(\vec{x}^{(i)})\Big)) \tag{4}$$

Once we determine the decision boundary, we can normalize the input data and run it through the algorithm to classify it to either side of the boundary.

## 2.2 K-Nearest Neighbors

The k-nearest neighbors classifier algorithm is a supervised machine learning technique based on the fact that ”similar” inputs exist close in proximity in the decision space. When classifying, we set the $k$ to be a constant of our choosing, which determines the number of neighbors we calculate the distance from in multidimensonal input space. From this boundary, we can group and classify the input data. The k-Nearest Neighbors algorithm is highly sensitive to outside noise, but can be very useful because of its efficient calculations. It holds massive predictive power, and gives a very simple and easy to interpret result, like most other classifiers.

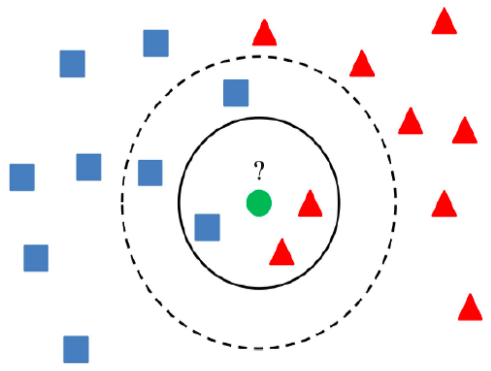

Figure 2: Graphical depiction of k-nearest neighbors classification

## 2.3 Multi-Layer Perceptron

An MLP is a class of artificial neural network, consisting of an input layer, hidden layers, and output layer. Each non-input node utilizes a $\tanh$ or ReLU activation function and backpropigation to train the network on the input data. It maps weighted inputs using the hidden layer to the output layer neuron. The network learns once the connection

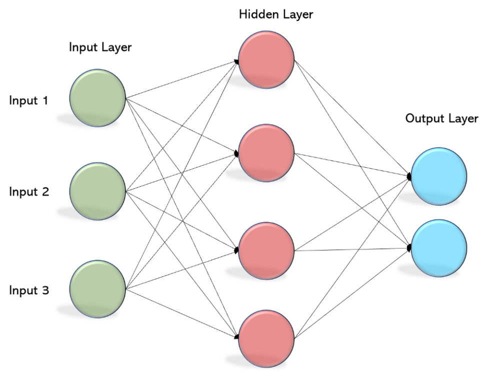

Figure 3: Graphical depiction of a multi-layer perceptron network

weights change after more and more data is processed. These weights minimize error by backpropigating the least squares algorithm, often using some form of gradient descent with an optimized learning rate. This can be used to find the optimal classification for 2 or more categories, all under the principle of minimizing the error of the weights on the algorithm.

## 2.4 Decision Tree

A classic decision tree can perform both classification and regression. In our case, we will use a classifying tree, which works as a chain of questions. The algorithm works as a series spanning from the root node, or original question. With

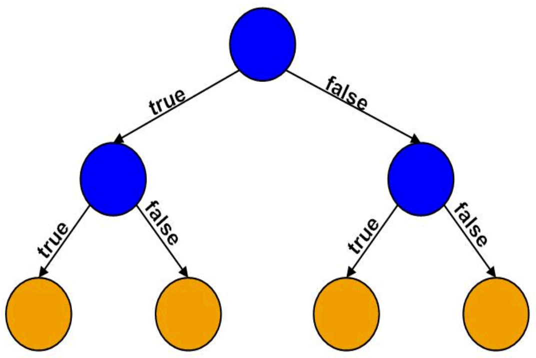

Figure 4: Graphical depiction of a generalized decision tree

each question, we split the dataset based off a specific feature and create new nodes. When we cap the nodes at a certain amount, the nodes end in leaves. Running a data point through the algorithm will send it to subsequent nodes based off of minimizing the entropy of each node, defined:

$$S = -\sum_{i=1}^{c} p_k \ln p_k \tag{5}$$

where $S$ signifies the entropy of each node, an $p_k$ represents the probability of picking a specific data point from the class $k$. Using the entropy loss function, a split at a node only goes through if the entropy defined from the split nodes is less than the entropy of the original parent node. Once we reach the leaves of the tree, we can group those nodes to overarchingly classify the data.

## 3 Dataset

Our dataset comes from a pkl file compiled of 440,091 insider trades from December 31st, 2003 until July 7th, 2021, which is all publicly accessible data complied by the SEC. The raw pkl file comes from S&P Global, and we made our own adjustments to the variables in order to choose which attributes we wished to analyze. We converted this pkl file into a dataframe, where each row makes up a single trade transacted in the time frame (440,091 rows $\times$ 46 columns), and there are 46 feature columns, giving a number of quantitative/qualatative information about each trade. The first columns of each trade give basic identification factors, such as the date and time of the transaction, ID number of whichever insider was making the trade, and the ticker symbol of the stock which was traded. Subsequent columns mark the type of transaction (acquisition or disposition), shares transacted/specific price, yield, market cap, etc. After running classification algorithms on the entire set of all trades and variables, we decided to downsample the dataframe, realizing that it was too large to quantify. We wrote our own simple downsampling algorithm to clean the data from 400,000 down to 100,000 trades. As far as variables, we then picked based off of intuition and curiosity which features we thought would have the most suggestive impact on *fwd_21d_returns*. We ended up choosing the following features to analyze, and created a new dataframe containing only such:

1. *percentOfSharesTraded*
2. *pct_change_in_shares_held_by_insider*
3. *n_shares_owned_post_transaction*
4. *beta_tv_median*, a measure of the ”risk” of a transaction
5. *marketcap*
6. *sector_values*
7. *transaction_values*
8. *cohort_names*, a specific grouping of trades based on the company’s background

From here, we implemented the machine learning techniques discussed in Section 2 to such features to predict our desired results.

## 4 Implementation

We used the four selected classification algorithms in order to classify binarily if a trade would bring about a positive forward return over the span of 21 days given a set of chosen quantitative and qualitative parameters. We obtained data from the public records of insider trades from 2004-2021, as explained before. After cleaning and downsampling, we created a method called *classifer* that would classify the forward 21-day returns as either a 1 or 0 (positive or negative) based on the sign of the column value for *fwd_21d_return*. We then added the four mentioned algorithms into a dictionary in order to run classification techniques on the data as follows using scikit-learn and a function called *classifiers_test*, where we imported sklearn metrics *train_test_split* and *accuracy_score* in order to split the dataset into the needed vector groupings and be able to calculate the desired results.

### 4.1 Logistic Regression

In implementing the logistic regression algorithm, we imported *sklearn.linear_model.LogisticRegression*, using all default penalty, stoppage, and strength criteria. We fit the split dataset vectors to the algorithm by using the *.fit()* feature, and chose not to look at the classification of individual points, but used the sklearn metric to print out the overall classification accuracy for all trades to an estimation of 2 decimal points. We created a short algorithm to introduce a mock investment strategy as well. We wrote *long_or_short* to read that if the predicted return is positive, we ”long” the stock (invest), and if negative, we short (sell). We then calculated the *returns* as the product of the *long_or_short* algorithm and the predicted *fwd_21d_returns*, and printed out the average return value for each prediction after standardizing the data.

### 4.2 K-Nearest Neighbors

To introduce k-Nearest Neighbors, we imported *sklearn.neighbors.KNeighborsClassifier*, using a default $k$ value of 5 and uniform weights for each feature. We fit the split dataset vectors to the algorithm by using the same algorithm as before, and once again use metrics to look at the overall accuracy of classification rather than predicting set points. Using *.fit()* we printed out the classification accuracy to an estimation of 2 decimal points, and the same *long_or_short* algorithm to calculate returns *returns* and average returns.

### 4.3 MLP Classifier

As for the Multi-Layer Perceptron, we imported *sklearn.neural_network.MLPClassifer*, using a default layer number of 100, ReLU activation function, and stochastic gradient descent to backpropagate the data. Once again, we fit the split dataset vectors to the algorithm by using the same algorithm and use metrics to look at the overall accuracy of classification rather than predicting set points. Using *.fit()* we printed out the classification accuracy to an estimation of 2 decimal points, and the same *long_or_short* algorithm to calculate returns *returns* and average returns.

### 4.4 Decision Tree

Lastly, for the Decision Tree, we imported *sklearn.tree.DecisionTreeClassifier*, using a entropy split at each node via the best option selection, with no limit on the tree depth. Once again, we fit the split dataset vectors to the algorithm by using the same algorithm and use metrics to look at the overall accuracy of classification rather than predicting set points. Using *.fit()* we printed out the classification accuracy to an estimation of 2 decimal points, and the same *long_or_short* algorithm to calculate returns *returns* and average returns.

## 5 Results & Further Research

As we will describe, each of the different algorithms worked with varying levels of accuracy:

1. (Top option) Decision Tree
   (a) $\approx 60\%$ accuracy $\implies \approx 1.5\%$ *fwd_21d_returns*
2. k-Nearest Neighbor
   (a) $\approx 53\%$ accuracy $\implies \approx +0.5\%$ *fwd_21d_returns*
3. Multi-Layer Perceptron
   (a) $\approx 50\%$ accuracy $\implies \approx 0\%$ *fwd_21d_returns*
4. (Worst Option) Logistic Regression
   (a) $\approx 50\%$ accuracy $\implies \approx 0\%$ *fwd_21d_returns*

The logistic classification worked fairly poorly, classifying 50,000 trades correctly only slightly better than by random chance. It also averaged negative returns on the initial investment over a 21- day period, which is obviously not optimal. We can attribute this to a fairly inefficient investment strategy, but the classifier overall compares poorly regardless. The k-Nearest Neighbor classifier worked slightly better, being able to correctly predict a trade's forward returns $53\%$ of the time. However, the average returns over 21 days barely break even in this case. Because the data set is so large and bound to have outliers, we can possibly attribute this to the fact that nearest neighbor algorithms are highly sensitive to noise, and may have worked better on a smaller dataset. The MLP classifier works similarly to the logistic classifier, only it breaks slightly even on returns, but does even worse classifying. The neural net may have been inefficient here, as 100 layers might have convoluted the data more than necessary, causing for the data to appear nothing more than random. Lastly, we found that the decision tree classifier worked extremely well. Able to correctly predict the sign of forward returns over $60\%$ of the time, it shows almost a $+1.5\%$ return on investment in only 21 days. Through the comparison of these different algorithms, we have easily concluded that the decision tree classifier could genuinely be implemented into an investment strategy. We feel that going forwards, we could efficiently use this classification to predict returns on our own investments. The results from the decision tree lead us to believe that there is definitely information to gain from looking at the transactions that insiders make, and hopefully use it to our own advantage. Especially because of the fact that we used a rudimentary long-short algorithm to buy or sell on a trade, using our classification technique via a Decision Tree with a more worked-through strategy could potentially bring massive returns.

## 6 Related Work

Through research, quantitative traders across the globe utilize machine learning and insider trades in a vast number of ways and with very similar data. One differentiating factor includes the databases' advanced algorithms to predict equity returns, in comparison to our simple long/short algorithm. Additionally, we found that some of these traders utilize the Russell 3000, a different mutual fund than the S&P 500, which we used. These Russell algorithms implement linear regression in R language to predict specific regression values instead of classification, but also have used Random Forest and recursion algorithms to do both regression and classification. They have seen much success, and open up the possibility of further applications of our decision tree to the Russell 3000 index.

## References

[1] T. Srivastava. Introduction to k-Nearest Neighbors: A powerful Machine Learning Algorithm (with implementation in Python & R). *Analytics Vidhya*, 2018.

[2] C. Bento. Decision Tree Classifier explained in real-life: picking a vacation destination. *Towards Data Science*, 2021.

[3] SEC. Insider Trade - Real-Time Buys. *gurufocus*, 2021.

[4] H. Wang. Sec11LogReg, Sec12ANN. In *MATH4570*, 2021.

[5] J. Ryle. Building an Insider Trading Database and Predicting Future Equity Returns. Northwestern University. *Northwestern University, Predictive Analytics, Students*, 2017.

[6] insider_trades.pkl. *S&P Global*, 2021.