Predicting Future Stock Price Directional Movements via Binary Classification Algorithms

Alex Greene, Lewis Wang, Charles Troutman, and Bryan Coronel

May 11, 2023

Abstract

Our project focuses on testing optimization methods for three machine learning models, SVM, logistic regression, and RNN/LSTM for predicting stock prices. We mainly explore various optimization methods for these models by training the models using different optimizers such as SMO for SVM or SGD/backpropagation through time for RNN/LSTM. We train/test these models on different stock/ETF datasets, but report results for APPL based on initial findings that APPL provided the best dataset for evaluating models/optimization methods. Our final results show that stock datasets are difficult to build accurate machine learning models for. SVM using SGD with an RBF kernel obtained the best results by a small margin with 0.532 accuracy, 0.63 ROC AUC, 0.526 F1 score, 0.552 precision, and 0.752 recall, although most models/optimization methods achieved similar results. We conclude that it is likely a challenging optimization problem to build machine learning models on noisy datasets like stock data.

1 Introduction

In recent years, predicting future stock price direction has become a significant problem in finance. It is a crucial task for investors and traders to make profitable investments. To address this issue, several machine learning algorithms have been developed to predict future stock price movements. Among other more complex models, there are fundamental algorithms such as logistic regression, support vector machine, and neural networks which have been widely used in financial applications.

Logistic regression is a statistical method used to analyze data with one or more independent variables that determine a binary outcome. Support vector machines are a type of supervised learning algorithm that can classify data into two categories via a variety of kernels depending on the application and characteristics of the feature space. Neural networks are a type of machine learning algorithm inspired by the structure and function of the human brain.

Nonlinear optimization algorithms are used to optimize the parameters of these models to achieve better performance. In this report, we applied different nonlinear optimization algorithms to models to see if their performance could be improved in predicting future stock price direction. The application of these models is not limited to stock price though. They have been used in various quantitative finance applications such as credit risk management, portfolio optimization, and fraud detection. Furthermore, the performance of these models has improved significantly over the years, mainly due to advances in machine learning algorithms and data availability.

However, it is essential to temper expectations when it comes to the accuracy of these models. It is important to remember that the future is uncertain, and no model can predict the future with 100% accuracy. Moreover, hedge funds exist for the sole purpose of predicting future stock price movements, and even they struggle to consistently outperform the market. Therefore, while these models can be useful tools, they should be used in conjunction with other methods and not relied on solely for investment decisions.

2 Implementation Details

2.1 Dataset

We ultimately chose to use the historical price data of NASDAQ: AAPL (Apple, Inc.), which contained the opening, closing, and daily high/low prices, in addition to the date and volume. When evaluating different datasets, the primary characteristic we focused on was the distribution of stock price movements, as stocks tend to go up over time. As a result, most models will detect this small, positive trend in the data, which could ultimately devolve into predicting a gain every time, regardless of the specific data in question. Hence, the first dataset we tried was the historical price data of SPY, which is the ETF of the S&P 500 Index (the 500 largest public companies in the US). Our thesis was that the linear trend would be least noticeable due to SPY's lower volatility. However, in practice, such low "noise" pushes the model to focus even more on the aforementioned trend.

Consequently, we then tried using a dataset where there was no linear trend. Hence, we modified the historical price data of NYSE: T (AT&T, Inc.) by only considering 1993 to 2023, which is when AT&T stock did not materially move (within 1% price difference) in either direction. Nonetheless, we did not see an improvement in the corresponding model, so we settled on Apple since it is a prominent company and had a good balance of volatility relative to AT&T and SPY.

In order to transform the dataset to become more suitable for binary classification, we created an additional column dubbed the "target," which was the daily close price shifted 5 days forward. We then created another column called "price direction," which compared the target price to the closing price 4 days before. If higher, then the directional value is 1 while, if lower, then 0. We also dropped the rows with missing target values so that we have no data inconsistencies.

2.2 Feature Selection

We added 6 additional features: (1) The daily "average" price, which was the average of the opening, closing, high, and low prices for each day; (2) The daily price percentage change, which was the percentage change from the open to closing prices; (3) The daily volume percentage change, which was the percentage change in volume from the previous day; (4) The daily "high-low" percentage, which is the percentage difference between the low and high prices of the day; (5) The relative strength index (RSI), which measures the "velocity" of price changes. It is calculated via: RSI = 100 - (100/ (1+RS)), where RS is the average gain when the stock is up divided by the average loss when the stock is down over 14 periods; and (6) MACD and Signal Line, which is the difference between the 26 day Exponential Moving Average from the 12 day Exponential Moving Average. The Signal line is the 9-day EMA of the MACD. We then dropped close and target, so those are not used anymore. We also broke out the price direction column into its own matrix so that we can separate X and Y. We also removed date and volume, as date does not make that much since as a variable and would lead to over fitting while volume is already reflected well in the feature engineering.

2.3 Training

Since we are using time series data, we used TimeSeriesSplit for cross-validation instead of randomly splitting the data into train and test groups. The latter approach results in data leakage because stock prices are not independent, so the model could be using the future to predict the past. As a result, TimeSeriesSplit divides the data into k cross folds while also maintaining temporal consistency. For example, if the dataset has 50 days of historical prices and we desire 5 folds of cross-validation, then the first fold would be days 1-10 for training and 11-50 for testing, then days 1-20 for training and 21-50 for testing, and so on. Because of this split, we also standardly scaled the data in training so that there is no data leakage there as well, as scaled data would reflect information about the whole distribution.

3 Evaluation

To evaluate the performance, we employed a variety of statistical metrics, including: (1) accuracy, the ratio between the correct predictions and total predictions; (2) precision, the ratio of true positives to true and false positives; (3) recall, the ratio of true positives to true positives and false negatives; (4) F1 score, which is the harmonic mean of precision and recall, helping to balance between precision and recall; (5) confusion matrix, which shows the true positives, true negatives, false positives, and false negatives counts; and (6) area under the ROC curve, which measures a classifier's ability to differ between the two classes, regardless of the classification threshold.

4 Support Vector Machines (SVMs)

4.1 Overview

The SVM classification algorithm was chosen due to its flexibility in handling complex data, such as financial data. Since historical stock data tends to be both noisy and non-linear, SVMs seems like a great choice, as maximizing the margin makes the model more robust to outliers and noise, while kernel functions like RBF can be used on non-linear data. Moreover, a common issue with stock prediction is over-fitting, yet the regularization parameter C can control the trade off between maximizing margin and minimizing the misclassification error. However, SVMs can become very computational expensive with a large dataset (either with more samples or features). We certainly experienced this case, as it took over 30 minutes for some of these models to train. Furthermore, the interoperability of SVMs is more constrained than say a simple multivariate regression, as the resulting model can be harder to understand, which makes further data analysis and additional improvements more difficult.

4.2 Specific Implementation Details

We used 3 different optimization methods: Sequential Minimal Optimization (SMO), Stochastic Gradient Descent (SGD), and Coordinate Descent (CD). Firstly, SMO is the "default" solver for many SVM packages because it breaks down the SVM into smaller "sub problems" that can be solved via Quadratic Programming, as they involve 2 Lagrangian multipliers. While this is great for small to medium datasets, there is meaningful performance degradation with larger and more complex datasets, like the one used in this project. Secondly, SGD approximates the solution by minimizing the hinge loss (a form of classification "loss"). While it can scale up well for large datasets, it requires careful hyperparameter tuning and experiences a slow convergence rate. Similarly, CD minimizes the objective function one "coordinate" at a time, meaning optimizing one parameter while keeping the rest constant. While this method can be applied to the dual problem, which allows updating the Lagrangian multipliers all at once, it can be slow to converge, where performance can depend on the update order or coordinate correlation, which can reduce the reproducibility of the results.

In addition to selecting 3 different optimization approaches, there were two further steps needed to implement SVMs. In addition to using the default linear kernel, we also used the radial basis function (RBF) kernel to obfuscate the inherent non-linearity in financial data. I opted for RBF over the polynomial kernel since RBF can combine multiple polynomial kernels at the same time and is generally more efficient. For hyperparameter tuning, we implemented gridsearch, which creating a discrete hyperplane of all possible hyperparameters, testing and then choosing the most optimal one. Our optimal hyperparameters:

Optimizer	**Optimal Hyperparameters**
CD	C=0.1
SMO	C=0.1, gamma=0.001
SGD	alpha=0.001, eta$_0$=1, max$_{iter}$=10000

4.3 Theory

Support Vector Machines aim to find the optimal hyper-plane that splits the binary classes into two separate parts, ideally where each part is composed of all the data points of one label. However, for data not linearly separable (like stock prices), then SVM can find the optimal hyperplane through kernels. Specifically, we are using support vector clustering, which uses SVMs to categorize unsupervised learning, mapping the data to these clusters in order to "train" our model.

Note: All formulas were directly from the references and not derived by us.

4.3.1 Linearly Separable Case and Margins

Linearly separable case is pretty intuitive, where there is a possibility for a hyperplane to separate the data. Let's assume the optimal hyperplane is π, then it can be expressed:

$$\pi = w^T \cdot x + b = 0$$

$$\pi^+ = w^T \cdot x + b = 1$$

$$\pi^- = w^T \cdot x + b = -1$$

Hence, our hard margin becomes $\frac{2}{||w||}$, so:

$$(w^*, b^*) = \text{argmax}_{w,b} \frac{2}{||w||} \text{ such that } y_i(w^T x_i + b) \geq 1$$

Soft margin operates the same way but introduces the concept of an "error function," in the sense that a parameter C can control the trade-off between the margin size and mis-classification error, thereby allowing for a "small" (relative to the value) amount of mis-classification. Hence, we can rewrite the constraint optimization as:

$$(w^*, b^*) = \text{argmax}_{w,b} \frac{2}{||w||} + C * \frac{1}{n} \sum_{i=1}^{n} \epsilon_i \text{ such that } y_i(w^T x_i + b) \geq 1 - \epsilon_i, \epsilon_i \geq 0$$

where mis-classified points are defined as $\epsilon_i > 0$ while classified points are $\epsilon_i = 0$. Thus, it is important to understand the impact of C. Generally, if C increases, then over-fitting increases - and vice versa (Stitson 1996).

4.3.2 Non-Linearly Separable Case and Kernels

However, if the data is not linearly-separable, then we can transform the data into a higher dimension and calculate the dot product. However, kernel functions are used to avoid these intensive calculations by mapping $R^n \to R^m$. If Φ is the mapping function, then the dot product is $\phi(x)^T \phi(y)$. Some of the common kernels include polynomial ($(1 + x_i^T x_j)^d$), RBF ($\exp(-\frac{||x_i - x_j||}{2\sigma^2})$), and Sigmoid ($\tanh(\gamma x_i^T x_j + r)$) (Stitson 1996).

4.3.3 The Lagrangian Connection: Primal/Dual Formulation

In order to solve our minimization problem, we can express the soft-margin classifier as

$$\min \frac{1}{n} \sum_{i=1}^{n} c_i \lambda ||w||^2 \text{ subject to } y_i(w^T x_i - b) \geq 1 - c_i \text{ and } c_i \geq 0$$

where the corresponding dual problem is:

$$\max \sum_{i=1}^{n} c_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i c_i (x_i^T x_j) y_i c_i \text{ subject to } \sum_{i=1}^{n} c_i y_i = 0 \text{ and } 0 \leq c_i \leq \frac{1}{2n\lambda}$$

This result is important because this becomes a quadratic programming problem, where w can be written as a linear combination of support vectors, which can be solved relatively easily compared to the primal (Lin). Note that we used the soft margin classifier due to the inherent imperfections in our dataset, as previously explained.

4.3.4 Coordinate Descent

One way to efficiently solve the optimization problem for Support Vector Machines (SVMs) is to use coordinate descent (Wright). This algorithm is attractive because it is simple to implement and can be more efficient than other methods, particularly when the function has a sparse or block structure.

In the case of SVMs, the objective function consists of a sum of convex and differentiable hinge loss and regularization terms. The hinge loss is a function of a single variable, while the regularization term is a function of all variables. This structure allows us to use coordinate descent to optimize each variable separately while keeping the others fixed.

Coordinate descent is an optimization algorithm that minimizes a function f by updating one coordinate at a time. It works for a function of the form $f(x) = g(x) + \sum_{i=1}^{n} h_i(x_i)$, where g is a convex, differentiable function and each h_i is a convex function of a single variable. At each iteration, the algorithm updates the value of one coordinate x_i while holding all others fixed. This update is performed for each coordinate in turn until convergence is reached.

The update for x_i is given by solving the one-dimensional minimization problem $\min_{x_i} f(x_1^{(k)}, \ldots, x_{i-1}^{(k)}, x_i, x_{i+1}^{(k-1)}, \ldots, x_n^{(k-1)})$. This update can be regarded as a descent in the direction e_i (or $-e_i$) where e_i is the ith unit vector. By sequentially minimizing with respect to different components, a relative minimum of f might ultimately be determined

This suggests that for $f(x)$, we can use coordinate descent to find a minimizer: start with some initial guess $x^{(0)}$, and repeat

$$\begin{aligned}
x_1^k &\in \operatorname{argmin} x_1 f(x_1, x_2^{(k-1)}, x_3^{(k-1)}, \ldots, x_n^{(k-1)}) \\
x_2^k &\in \operatorname{argmin} x_2 f(x_1^{(k)}, x_2, x_3^{(k-1)}, \ldots, x_n^{(k-1)}) \\
x_3^k &\in \operatorname{argmin} x_3 f(x_1^{(k)}, x_2^{(k)}, x_3, \ldots, x_n^{(k-1)}) \qquad \vdots \\
x_n^k &\in \operatorname{argmin} x_n f(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \ldots, x_n)
\end{aligned}$$

for $k = 1, 2, 3, \ldots$ Note: after we solve for $x_i^{(k)}$, we use its new value from then on.

While coordinate descent is a powerful optimization technique, it may not converge to a global minimum, and its convergence properties can be poorer than gradient descent. Nonetheless, it remains an important tool in the SVM toolbox due to its efficiency and simplicity.

4.3.5 Sequential Minimal Optimization (SMO)

SMO is a special application of the coordinate ascent method for the SVM dual optimization problem (1), which is the dual formulation of the primal SVM optimization problem. The objective of the SMO algorithm is to minimize the dual objective function.

Recall the KKT complementary slackness condition. Given the general optimization problem

$$\min_{x \in R^n} f(x) \quad \text{subject to} \quad h_i(x) \leq 0,\ i = 1, \ldots, m, \quad g_j(x) = 0,\ j = 1, \ldots, r,$$

the Karush-Kuhn-Tucker (KKT) complementary slackness condition is satisfied when $u_i \cdot h_i(x) = 0$ for all i (complementary slackness)

This condition is used to identify the variables to optimize in each SMO iteration. Specifically, the SMO algorithm works by iteratively selecting a pair of α_i and α_j that violate KKT complementary slackness, and optimizing the dual objective function with respect to those two variables while holding all others fixed. This process continues until convergence, which is determined by checking if KKT complementary slackness is satisfied within a certain tolerance level (usually around 10^{-3}).

It's also worth noting that the SMO algorithm is a special case of coordinate descent. The pair of α_i and α_j that violate the KKT complementary conditions used to optimize the dual objective function with respect to those two variables can be seen as updating the i-th and j-th coordinates of the vector α. Overall, SMO is an efficient optimization algorithm for training linear SVMs, and well handle large datasets with high-dimensional feature spaces.

4.4 Results

Note: All data is in the appendix to reduce paper length.

For CD, in 5 folds of cross-validation, only $\frac{2}{5}$ models had an accuracy greater than 0.5, which is the random chance benchmark. Hence, the model generally performed worse than if randomly guessing. However, the strong variation in accuracy (ranging from 43.6% to 52.2%) implies that the model is not sufficiently robust. Furthermore, the ROC AUC Score follows a similar pattern, where all the scores are but one are slightly above 0.5, again implying that the model's performance is slightly better than random chance, even if the accuracy is worse. All but one precision metric (the lowest being 0.490) indicate that the model can correctly predict stock gains a little more than half the time, but the corresponding low recall (especially on the last fold being 0.174) demonstrates that the model struggles to even identify gains, despite its accuracy on it. Additionally, the confusion matrix shows the model's conservative bias, as it often underestimates the chance of gains given that there is a higher false negative than false positive count. As a result, the F1 scores experience large variation (from 0.264 to 0.463), similar to accuracy, which which highlights the generally inconsistency of the model as it cannot accurately balance between false positives and false negatives. Unfortunately, SMO is a worse approach than CD. The 1.0 recall, all future predictions being gain, and a lopsided confusion matrix are all indicative of the fact that this model just predicted gain every single time, regardless of the data present. SGD is not predicting all gains or all losses, which is an improvement over SMO. Furthermore, the main difference between the two kernels is in recall, as the RBF kernel had a 0.752 average recall while the Linear kernel had a 0.631 average recall. This disparity flows through the confusion matrices, where the split of true/false positives/negatives are not equal. Clearly, this eventually affects future predictions, where in CV=3, RBF predicts all losses but linear predicts four losses and one gain. This is not surprising, as RBF is supposed to perform better with non-linear data. Moreover, the average F1 score is actually higher in the RBF kernel (0.526 vs. 0.524), implying that, though the RBF is more "biased," it does a better job of accurately balancing the distribution of gain/loss predictions. While the recall difference does not materially affect accuracy or F1, it is generally better to have a model that weighs the "chance" of a price movement somewhat equally, as that case is more true in the short run than the long run, which is when a trading strategy based on this research would be implemented. Nonetheless, regardless of kernel choice, all the accuracies and ROC AUC scores are greater than $\frac{1}{2}$, demonstrating that this model performs a little bit better than randomly guessing.

4.5 Discussion

Method	Accuracy	RAS	F1 Score	Precision	Recall	Confusion Matrix	Future Predictions
CD	0.483	0.339	0.509	0.536	0.327	[[508.2, 250.6], [600.8, 288.4]]	Mixed
SMO$_{RBF}$	0.540	0.700	0.497	0.540	1.0	[[0, 758.8], [0, 889.2]]	All Gain
SMO$_{Linear}$	0.540	0.700	0.498	0.540	1.0	[[0, 758.8], [0, 889.2]]	All Gain
SGD$_{RBF}$	0.532	0.630	0.526	0.552	0.752	[[209.6, 549.2], [215.8, 667.4]]	Mixed
SGD$_{Linear}$	0.532	0.590	0.524	0.554	0.631	[[311.2, 447.6], [323.8, 565.4]]	Mixed

When evaluating all models holistically, the one with the highest accuracy was SMO; however, this is because both models predicted gain regardless of any other information. And, since stocks tend to go up more than they go down, the subsequent performance was the highest. However, that is not super useful, so when focusing on the remaining two models, it is clear that SGD has higher accuracy, area under the ROC curve, F1 score, precision, and recall than CD (where all but the last tend to point towards better performance). Not only that, but CD tends to focus more on "loss" than "gain," which is not desired given the lower presence of losses over time.

Therefore, the optimal model is SGD$_{RBF}$ because it has a higher area under the ROC curve score (0.630 vs. 0.590) than the Linear kernel, yet still has the same accuracy. Furthermore, the RBF kernel has a marginally higher F1 score but also has a marginally lower precision than Linear. As a result, the RBF model has a higher recall, but we believe this is due to its ability to handle non-linear data and the general presence of more gains than losses. As a result, even though the confusion matrix seems to be less even and there is a higher recall, SGD$_{RBF}$ seems like the optimal model.

5 Logistic Regression

5.1 Overview

Logistic regression has a long history of applications in finance, including stock price prediction. It can model the probability of a stock price movement as a function of various independent variables, such as historical prices, trading volumes, and other financial indicators (Correa-Alvarez 2022). By analyzing the coefficients of the model, we can gain insights into which factors are most strongly associated with stock price movements. Compared to other classification algorithms such as SVM and neural networks, logistic regression is less computationally expensive and more interpretable, making it suitable for problems with fewer input features. In contrast to SVM, logistic regression provides a simpler and more interpretable model that can be efficiently tuned to avoid overfitting by using regularization techniques. Logistic regression can also perform better than neural networks when there is a limited amount of data available, as it requires fewer parameters to estimate. To make logistic regression work well with financial data, feature selection techniques can be used to identify the most relevant variables, reducing the risk of overfitting. Moreover, domain-specific knowledge of finance can be utilized to create informative and meaningful features that enhance the model's performance (Agnieszka Strzeleck 2020). This section of the report will touch on methods to initialize hyperparameters, select features, and optimize combinations of hyperparameters.

5.2 Specific Implementation Details

For our binary classification problem, we implemented a logistic regression model using scikit-learn's LogisticRegression class. After feature engineering previously mentioned, using TimeSeriesSplit, and performing cross-validation, we optimized our logistic regression model using multiple techniques. First, we used basin-hopping to optimize the initial value of the "C" hyperparameter. Next, we used particle swarm optimization to select the best subset of features to use for the model. This step would have helped us eliminate irrelevant features that could negatively impact the performance of the model (Meng 2020). However, it turned out that after features we had initially preprocessed were sufficiently relevant. After selecting the features, we used a random search to find the best bounds for different hyperparameters, including penalty, C, solver, max_iter, and fit_intercept. Finally, we plugged the best hyperparameters into grid search to get the best possible solution. We scaled our input features using StandardScaler to ensure that they were on the same scale. The best hyperparameters were chosen based on accuracy as the scoring metric.

5.3 Theory

Suppose we have a dataset of n observations, where the ith observation has a binary response variable y_i and a vector of predictor variables $\mathbf{x}i = (xi1, x_{i2}, ..., x_{ip})^T$. We assume that the conditional probability of y_i given $\mathbf{x}_i$ follows a Bernoulli distribution with parameter π_i, which is modeled using the logistic function:

$$\pi_i = P(y_i = 1|\mathbf{x}_i) = \frac{1}{1+\exp(-\mathbf{x}_i^T\boldsymbol{\beta})} \tag{1}$$

where $\boldsymbol{\beta} = (\beta_0, \beta_1, ..., \beta_p)^T$ is a vector of parameters to be estimated, and β_0 is the intercept term.

The likelihood function for the logistic regression model is:

$$\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} \pi_i^{y_i}(1-\pi_i)^{1-y_i} \tag{2}$$

The log-likelihood function is:

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} y_i \log(\pi_i) + (1-y_i)\log(1-\pi_i) \tag{3}$$

The goal is to find the maximum likelihood estimates of the parameters $\boldsymbol{\beta}$, which maximize the log-likelihood function. This is a nonlinear optimization problem, as the log-likelihood function is typically non-convex and has many local maxima.

We can use gradient ascent or gradient descent to iteratively update the parameter estimates in the direction of the gradient of the log-likelihood function, until convergence. The gradient of the log-likelihood function with respect to the parameter β_j is:

$$\frac{\partial \ell(\boldsymbol{\beta})}{\partial \beta_j} = \sum_{i=1}^{n}(y_i - \pi_i)x_{ij} \tag{4}$$

Using the gradient, we can update the parameter estimates as follows:

$$\beta_j^{(t+1)} = \beta_j^{(t)} + \alpha \frac{\partial \ell(\boldsymbol{\beta})}{\partial \beta_j} \tag{5}$$

where α is the learning rate, and t is the iteration number.

To ensure convergence and prevent overfitting, regularization terms can be added to the log-likelihood function. The two most common regularization methods used in logistic regression are L1 regularization (lasso) and L2 regularization (ridge). L1 regularization adds a penalty term proportional to the absolute value of the parameter estimates, while L2 regularization adds a penalty term proportional to the square of the parameter estimates.

The regularized log-likelihood function is:

$$\ell_{reg}(\boldsymbol{\beta}) = \ell(\boldsymbol{\beta}) - \lambda_1 \sum_{j=1}^{p} |\beta_j| - \frac{\lambda_2}{2} \sum_{j=1}^{p} \beta_j^2 \tag{6}$$

where λ_1 and λ_2 are regularization parameters that control the strength of the regularization (Febrianti 2021).

The gradient and Hessian of the regularized log-likelihood function can be derived similarly, and optimization algorithms such as gradient descent, Newton's method, and BFGS can be used to estimate the regularized logistic regression parameters.

In summary, the estimation of parameters for a logistic regression model involves nonlinear optimization techniques, such as gradient descent, Newton's method, and BFGS, to maximize the log-likelihood function. Regularization methods can also be used to prevent overfitting, and the regularized log-likelihood function can be optimized using similar nonlinear optimization techniques.

5.3.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population-based stochastic optimization algorithm that has been widely used to solve various optimization problems in engineering, finance, and other fields (El Dor et al. 2012). The algorithm mimics the social behavior of birds flocking or fish schooling, where individuals interact and learn from one another to achieve a common goal.

In Particle Swarm Optimization (PSO), candidate solutions are represented by particles, which traverse the search space to find an optimal solution by interacting and sharing information with neighboring particles. Each particle maintains its individual best solution (local best) and computes the neighborhood best. The global best solution obtained in the entire swarm is updated at each step of the algorithm. Using this information, particles identify successful locations in the search space and are guided by these successes. In each step of the algorithm, a fitness function is used to evaluate particle success.

Algorithm 1: Symmetric Chain Decomposition (SCD) Algorithm

```
Input  : Let $n$ be the desired number of vertices for the poset, $m$ be the maximum rank size, and $k$ be the maximum
         number of ranks.
Output: Symmetric Chain Decomposition (SCD) of the poset.
Initialize a uniformly random symmetric and unimodal rank scheme $r = (r_1, r_2, \ldots, r_k)$, where $r_i \in 1, 2, \ldots, m$ and
  $r_i \leq r_{i+1}$ for all $i$; Randomly place $n$ vertices on ranks according to the rank scheme $r$;
while the resulting poset is not strongly Sperner do
    Randomly add an edge between two incomparable elements;
Choose the middle three ranks $A, B$, and $C$; Create two bipartite graphs $G_1 = (A, B, E_1)$ and $G_2 = (B, C, E_2)$, where
  $E_1$ and $E_2$ are the set of incomparable pairs between $A$ and $B$, and between $B$ and $C$, respectively; Let
  $t = \min(|E_1|, |E_2|)$, and choose one of $G_1$ or $G_2$ to be the "accommodator" graph and the other to be the
  "to-accommodate" graph. If $|E_1| = |E_2|$, choose one randomly;
for each perfect matching $M$ of size $t$ in the accommodator graph do
    Find a random perfect matching $P$ of size $t$ for the vertices in $G_1$ or $G_2$ (whichever is the to-accommodate graph);
      Attempt to align $P$ and $M$ by constructing a bipartite graph $G = (P, M, E)$, where $E$ is the set of pairs $(u, v)$
      where $u \in P$ and $v \in M$ are comparable in the poset; if $G$ contains a perfect matching then
        Contract the three ranks $A, B$, and $C$ into one, using the pairs in the perfect matching as the symmetric chains;
          break;
Repeat steps 4-8 with a random number $t'$ of edges chosen uniformly at random between 1 and $t$. If no SCD is found
  after $k'$ iterations, make a probabilistic statement that there's a high chance that there is no SCD of the poset;
Choose the new middle three ranks and repeat steps 4-9 until the entire poset is contracted into one symmetric chain.
  The resulting set of symmetric chains is the SCD of the poset;
```

To model the swarm, in iteration t, each particle n moves in a multidimensional space according to its position $\mathbf{x}n^{(t)}$ and velocity $\mathbf{v}n^{(t)}$ values, which are highly dependent on its local best $\mathbf{v}1n^{(t)}$, also known as the cognitive component, and the global best $\mathbf{v}2^{(t)}$, typically known as the social component. The velocity update equation is given by:

$v_n(t+1) = wv_n(t) + c_1r_1(p_{1n} - x_n(t)) + c_2r_2(p_2 - x_n(t))$

where w, c_1, and c_2 are coefficients that assign weights to the inertial influence, the local best, and the global best when determining the new velocity $\mathbf{v}_n^{(t+1)}$, respectively. Typically, the inertial influence is set to a value slightly less than 1. r_1 and r_2 are random vectors with each component generally a uniform random number between 0 and 1, intended to multiply a new random component per velocity dimension, rather than multiplying the same component with each particle's velocity dimension. The particle's position update equation is given by:

$x_n(t+1) = x_n(t) + v_n(t+1)$

In PSO, the particles' trajectories are influenced by their previous direction, the best location they personally found thus far, and the best location the flock collectively found thus far.

Particle Swarm Optimization (PSO) is less computationally intensive than other optimization methods due to its simple and efficient search strategy. Unlike other methods that require complex mathematical models or extensive problem-specific knowledge, PSO relies on a simple update rule and the collective behavior of particles to explore and exploit the search space. This makes it particularly effective for high-dimensional and complex optimization problems, where other methods may struggle due to the curse of dimensionality.

5.3.2 Basin-Hopping

The basin-hopping algorithm is a stochastic global optimization algorithm designed to find the global minimum of an objective function (Vinkó, T., Gelle, K. 2017). It was originally developed to address the challenge of finding stable molecular configurations or configurations with the lowest energy in Physical Chemistry, where many local minima can make it difficult for standard optimization methods that depend heavily on initial conditions. In the basin hopping algorithm, an initial state is generated and local optimization is performed on it. Then, the algorithm randomly perturbs the solution and performs local optimization again. If the new solution is accepted, it becomes the new starting point; otherwise, the previous solution is retained. This process continues until the stopping criterion is met. The following is the pseudocode for the basin-hopping algorithm.

Algorithm 2: Basin Hopping Algorithm based off of that of Vinkó, T., Gelle, K. (2017)

```
Data: Objective function $f$, random perturbation function randomPerturbation, initial temperature $T_0$, cooling
      schedule $T(t)$, and stopping criterion for the sequence of states $X_j j = 0^n$
Result: Optimal state $X$opt
Initialize $X_0 \leftarrow$ randomState(); $n \leftarrow 0$;
while stopping criterion for $X_j j = 0^n$ not satisfied do
    $Y_n \leftarrow$ randomPerturbation($X_n$);
    $U_n \leftarrow$ LocalMinimization($Y_n$); $V \sim$ Uniform$[0, 1]$;
    if $V < \min 1, \exp[-(f(U_n) - f(X_n))/T(n)]$ then
        $Xn + 1 \leftarrow U_n$;
    else
        $X_{n+1} \leftarrow X_n$;
    $n \leftarrow n + 1$;
    $T(n) \leftarrow$ cooling schedule($T_0, n$);
$X_{\text{opt}} \leftarrow \arg\min_{X_j \in X_0, X_1, \ldots, X_n} f(X_j)$;
```

The basin hopping algorithm's main concept is to enhance local deterministic optimization by incorporating a random perturbation step that can evade local minima. To elaborate, during the `RandomPerturbation` process, a random perturbation vector $W \in R^d$ is generated and combined with the current state X_n to produce a new state $Y_n = X_n + W$. The perturbation vector W is typically spherically symmetric or has independent coordinates. The state Y_n serves as the starting point of a deterministic local minimization procedure.

After the choice LocalMinimization procedure, the resulting local minimum U_n is then either accepted or rejected as the new state with probability equal to

$$\min 1, \exp\left[-\frac{f(U_n) - f(X_n)}{T}\right],$$

where $T \geq 0$ is a fixed temperature parameter. This means that downward steps for which $f(U_n) < f(X_n)$ are always accepted. The BH algorithm repeats this basic step until a predefined stopping criterion is satisfied.

Commonly used stopping criteria for the BH algorithm include a limit on the number of evaluations of the function f or the absence of improvement over several consecutive iterations. The monotonic basin hopping method is the BH variant corresponding to the limiting case $T = 0$, in which all steps that increase the energy are rejected.

Basin hopping can thus be viewed as a random walk on the set of local minima of the energy landscape, which because its transition probabilities favor downhill moves to lower minima, is capable of finding the global minimum and, hence, of solving global optimization problems. Its transition probabilities depend in a complex way on the current position, the landscape, and the perturbation step.

The stochastic nature of basin hopping can help avoid getting stuck in local minima, leading to more robust and reliable results. Basin hopping can also be easily parallelized, making it well-suited for high-performance computing environments.

5.4 Results

Recall, I started by tuning a base logistic regression model and recorded its performance metrics. After using basin-hopping to optimize the initial value of the "C" hyperparameter, we measured the performance again. Then, I employed particle swarm optimization to select the best subset of features to use for the model and re-evaluated. This allowed for random search to find bounds for different hyperparameters. Finally, I employed grid search to obtain the best solution possible for the model's hyperparameters in a more efficient manner based on bounds near values yielded from Random Search. This best-performing model achieved an accuracy, on average across folds, of 0.524, an F1 score of 0.495, and a ROC AUC score of 0.526. The precision score was 0.519, and the recall score was 0.520, indicating that the model can decently balance the two classification labels. It is worth noting that all models had an accuracy and ROC AUC score greater than 0.5, demonstrating that they performed better than randomly guessing but not by much. Nonetheless, the best model outperformed the other models, and its performance metrics were close to those of the SVM model described in my teammate's section, although there were some differences in the recall score between the two models. The model's performance in predicting future price movements improved significantly from the base logistic regression to the final bounded grid search model. Notably, the model's predictions became more realistic by not solely predicting one direction of price movement, but rather a mix of both gains and losses. This positive development indicates that the model is taking a more balanced approach to price movement, which is generally more realistic in the short run. Overall, though, it is worth cautioning about using our relatively simple models in the grand scheme of the quantitative finance industry. In the long run, other methods of prediction would likely be better.

Results after each iteration of optimization:

Method	Accuracy	F1 Score	ROC AUC	Precision	Recall	Future Predictions
Base Log Reg	0.511	0.579	0.511	0.543	0.620	All Gain
Basin-hopping Initialization	0.512	0.579	0.523	0.543	0.622	All Gain
PSO Feature Selection	0.512	0.579	0.523	0.543	0.622	Mixed
Random Search	0.512	0.608	0.550	0.570	0.652	Mixed
Bounded Grid Search	0.524	0.495	0.526	0.558	0.520	Mixed

Cross-validation results of the final model:

CV	Accuracy	F1 Score	ROC AUC	Precision	Recall	Future Predictions
1	0.513	0.166	0.540	0.519	0.100	[G, L, G, L, G]
2	0.528	0.545	0.545	0.525	0.566	[G, G, G, G, G]
3	0.505	0.569	0.472	0.545	0.595	[G, L, L, G, G]
4	0.520	0.522	0.557	0.614	0.453	[G, L, G, L, G]
5	0.556	0.673	0.515	0.589	0.787	[G, G, L, G, G]
Avg	**0.524**	**0.495**	**0.526**	**0.558**	**0.520**	
Std	**0.018**	**0.196**	**0.031**	**0.037**	**0.261**	

5.5 Discussion

Upon evaluating all incremental tuning steps, the model with the best performance was obtained after performing all the tuning steps. The base logistic regression model initially had low accuracy and other metrics, but tuning the "C" hyperparameter with basinhopping resulted in improved accuracy and other metrics. This better starting point for "C" may have allowed the model to find more optimal solutions faster. Additionally, the performance was further improved after using particle swarm optimization to select the best subset of features and random search to determine the bounds of different hyperparameters. This could have eliminated redundant or irrelevant features, reducing overfitting and improving generalization. Finally, plugging the best parameters into grid search yielded the optimal model with the best performance in terms of accuracy, precision, recall, and F1 score. Searching over a larger space of hyperparameters may have allowed the model to find more optimal solutions that it wouldn't have otherwise. The incremental tuning process helped to identify the best combination of hyperparameters and features, resulting in a more accurate and reliable model. Each step in the optimization process could have contributed to the model's improvement in different ways, from finding a better starting point to reducing overfitting to exploring a wider range of solutions to fine-tuning the model for optimal performance. By incrementally tuning your model, we were able to iteratively improve its performance and achieve the best results possible. Overall, the incremental tuning process was a key factor in achieving the best possible model performance.

6 Recurrent Neural Networks/Long Short-term Memory

6.1 Overview

Recurrent neural networks (RNN) are powerful machine learning models ideally suited for handling time series data due to outputs from nodes being able to affect later inputs to those nodes. While other machine learning models such as other neural network architectures assume that each data point is independent from other data points, recurrent neural networks instead attempt to find patterns through time among time series data. This works by reusing previous outputs from a node as parts of future inputs. This ensures that patterns through time among the time series data are captured by the recurrent neural network.

A specific type of RNN is Long Short-term Memory (LSTM) which allows for longer term patterns in data to be captured and stored by the LSTM (Shertinsky 2021). This works by adding another component to nodes which determine when to remember and when to forget past data. If certain aspects of past data are important, they may be remembered for longer periods of time compared to other aspects of past data which may be forgotten. While a traditional RNN may also be able to capture long term trends, this may not be computationally feasible since it would have to remember all past data since the RNN would not learn when to forget past data.

6.2 Specific Implementation Details

For our RNN/LSTM implementation, we used 5 LSTM layers, including 1 input layer and 4 hidden layers. Each layer has 50 nodes. The input data for the input layer is the series of 5 days of data preceding a target prediction for a day's price movement. Details on the data features and target are found in the Implementation Details section earlier in the paper. We used a series of 5 days of data, which means 5 timesteps for the LSTM, as input to the LSTM to better capture trends for 5 days before a given prediction. We also experimented with larger timesteps such as 50 days but this was too computationally expensive and did not result in a significant increase in accuracy so we used 5 days for our final results. We also applied 0.5 dropout to all LSTM layers to prevent overfitting on the training data since the validation data for each fold can vary greatly from the training data. This is because the historical prices for a single stock like Apple are fairly volatile between different periods. There could be a strong upward trend in one time period and a strong downward trend in the following time period. The final output layer has one node with a sigmoid activation which is useful for binary classification of days as increasing or decreasing in price. Training was performed for 50 epochs for each optimizer/learning rate. We tested Adam/SGD with various learning rates. More details on the optimization process of a RNN/LSTM are described in the section below.

6.3 Theory

In terms of theory, LSTM was developed due to issues with traditional RNN models in capturing data/trends over longer periods of time (Hochreiter and Schmidhuber 1997). In their paper proposing LSTM, the authors describe how older RNN models struggled with determining what data to put in short-term memory, meaning what a model should remember when considering future data. LSTM uses a gradient-based method to determine the states of units that control what data is remembered/forgotten. Error from the gradient-based method is stored in the LSTM units until they learn to forget that data. This allows LSTM to learn when/how long to remember data and allows for longer term patterns in data to be learned. This is unlike traditional RNN where long-term patterns are difficult computationally and practically for a model to learn.

6.4 Optimization

6.4.1 SGD

Stochastic Gradient Descent is a probabilistic approximation of Gradient Descent used for optimizing the weights of a neural network (Roberts et al. 2022). It is an approximation because, at each step, the algorithm calculates the gradient for one observation picked at random, instead of calculating the gradient for the entire dataset.

Consider a neural network with initial weights and biases given by $w^{(l)}(0)$ and $b^{(l)}(0)$, respectively. Let $\eta > 0$ be the learning rate. The SGD update rule for step $n + 1$ is given by the following recursive sequence:

$$w^{(l)}(n+1) = w^{(l)}(n) - \eta\delta^{(l)}(n)x^{(l-1)}(n)ij \; b^{(l)}(n+1) \qquad = b^{(l)}(n) + \eta\delta^{(l)}(n)j,$$

where $x^{(l)}(n)$ and $\delta^{(l)}(n)$ depend on the weights and biases obtained at the nth step. The gradient $\delta^{(l)}(n)$ is calculated using the derivative of the activation function, ϕ', and the cost function, C. The size of the gradient depends on the type of activation function used and the choice of cost function.

It is important to choose an appropriate learning rate, η, as a rate that is too large can lead to the algorithm not converging, while a rate that is too small will result in a longer convergence time and possible local minima. For deep feed-forward neural networks, ReLU activation functions are most commonly employed to avoid the saturation tendency of sigmoidal functions.

6.4.2 Adam

ADAM (Adaptive Moment Estimation) is an optimization algorithm used in deep learning to update the weights of neural networks during training. It combines the benefits of two other popular optimization algorithms, AdaGrad and RMSProp.

$$v(t) = (1-\gamma)\sum_{i=1}^{t}\gamma^{t-i}x_i^2 < M^2(1-\gamma) = M^2(1-\gamma^t)$$

This adaptive learning method was inspired by the previous AdaGrad and RMSProp methods and was formally introduced in 2014 (Kingma and Ba 2015). The method uses the estimation of the first and second moment of the gradient by exponential moving averages and then applies some bias corrections.

The cost function, $C(x)$, subject to minimization, may have some stochasticity built in and is assumed to be differentiable with respect to x. We are interested in the minimum $x^* = \operatorname{argmin}_x E[C(x)]$.

For this, we denote the gradient of the cost function at the time step t by $g_t = \nabla C(x(t))$. We consider two exponential decay rates for the moment estimates, $\beta_1, \beta_2 \in [0, 1)$, fix an initial vector $x(0) = x_0$, initialize the moments by $m(0) = 0$ and $v(0) = 0$, and consider the moments updates

$$m(t) = \beta_1 m(t-1) + (1-\beta_1)g_t \; v(t) \qquad = \beta_2 v(t-1) + (1-\beta_2)(g_t)^2,$$

where $(g_t)^2$ denotes the elementwise square of the vector g_t. The moments $m(t)$ and $v(t)$ can be interpreted as biased estimates of the first moment (mean) and second moment (uncentered variance) of the gradient given by exponential moving averages. These estimates are biased due to initialization and decay factors. The bias correction is applied by computing the moments recursively using the decay factors. The resulting bias-corrected moments are then used to update the weights of the neural network. ADAM has been shown to perform well on sparse gradients and noisy data, and is widely used in deep learning applications.

The Adam algorithm combines the advantages of both AdaGrad and RMSProp. It performs well on sparse gradients and noisy data, and has been shown to be effective in a wide range of deep learning applications.

6.4.3 Backpropagation Through Time

Backpropagation through time (BPTT) is a variant of the standard backpropagation algorithm used to train recurrent neural networks (RNNs) (Roberts et al. 2022). It is unique to RNNs because they process input sequences of variable length and maintain an internal state that depends on the previous inputs. BPTT computes the gradients of the loss function with respect to the RNN's parameters by unfolding the network through time and applying the standard backpropagation algorithm to the resulting feedforward network. This allows the gradients to be propagated through the entire sequence and back into the RNN's hidden state, allowing the network to learn long-term dependencies in the input sequence.

Recall that Backpropagation is a procedure used to compute the gradient of a cost function $C(w, b)$ with respect to the weights and biases of a neural network. The cost function measures the proximity between the network output $Y = f(w, b, X)$ and the target Z. The gradient of C is given by $\nabla C = (\nabla_w C, \nabla_b C)$, where $\nabla_w C$ and $\nabla_b C$ are the gradients of C with respect to the weights w and biases b, respectively. These gradients are then used to update the weights and biases using an optimization algorithm such as SGD.

Backpropagation works by computing the gradient of C with respect to the output Y, and then recursively applying the chain rule to compute the gradient of C with respect to each layer of the network. The gradient of C with respect to Y is given by $\nabla_Y C = \nabla_C^T (\partial Y/\partial Z)$, where ^{T} denotes the transpose operator and $(\partial Y/\partial Z)$ is the Jacobian of the output Y with respect to the target Z.

To compute the gradient of C with respect to a layer l, we first compute the gradient of C with respect to the output of layer l, denoted by ∇Y_l. Then, we use the chain rule to compute the gradient of C with respect to the input of layer l, denoted by ∇I_l, as $\nabla I_l = (\partial Y_l/\partial I_l)^T \nabla Y_l$. Finally, we use the gradient of C with respect to the input of layer l to compute the gradients of C with respect to the weights and biases of layer l as $\nabla w_l C = x_{l-1}^T \nabla I_l$ and $\nabla b_l C = \nabla I_l$.

In the case of an RNN without bias parameters, where the activation function in the hidden layer uses the identity mapping, the input, hidden state, and output at time step t are denoted by x_t, h_t, and o_t, respectively. The weight parameters are denoted by W_{hx}, W_{hh}, and W_{qh}. The loss at time step t is denoted by $l(o_t, y_t)$. The objective function, the loss over T time steps from the beginning of the sequence, is given by the sum of the losses at each time step, i.e., $C = \sum_t l(o_t, y_t)$. The gradients of C with respect to the weight parameters are computed using backpropagation through time. To minimize the loss function using

To minimize the loss function using gradient descent, we iteratively update the values of the weights and biases in the network in the opposite direction of the gradient of the loss function with respect to those parameters. This process is repeated multiple times until the loss function converges to a minimum or until a certain stopping criteria is met. During each iteration of gradient descent, we calculate the gradient of the loss function with respect to the parameters of the network using the chain rule of calculus. The gradient is then used to update the values of the weights and biases in the network by taking a step in the opposite direction of the gradient, scaled by a learning rate hyperparameter.

There are several variations of gradient descent, such as stochastic gradient descent (SGD), mini-batch gradient descent, and Adam optimizer. These methods differ in how they update the weights and biases and how they estimate the gradient of the loss function. Choosing the appropriate optimization method is important for achieving good performance in deep learning models.

6.5 Results/Discussion

Detailed cross-validation results for our RNN/LSTM optimized with Adam/SGD using varying learning rates are displayed in the appendix. The overall results which average each cross-validation fold are displayed in the table below. Overall RNN/LSTM Results:

Method	Accuracy	ROC AUC	F1 Score	Precision	Recall	Confusion Matrix
Adam (0.001 lr)	0.524	0.526	0.495	0.558	0.500	[[2048, 1743], [2175, 2292]]
Adam (0.0001 lr)	0.518	0.518	0.492	0.560	0.475	[[2129, 1662], [2311, 2133]]
SGD (0.001 lr)	0.524	0.504	0.294	0.232	0.400	[[2407, 1384], [2534, 1910]]
SGD (0.01 lr)	0.524	0.511	0.294	0.232	0.400	[[2407, 1384], [2534, 1910]]

The table above shows the average results across 5 folds of data, each representing 5 years of Apple stock prices. For each fold, training was performed on all data preceding the validation data for each fold. Based on the table above, we observe that Adam with an 0.001 initial learning rate appears to perform better than the other optimizers for optimizing our LSTM model. As seen, the accuracy of 0.524 is tied for the highest and the ROC AUC, F1 score, and recall are the highest while the precision is near the highest as well. The SGD optimizer, with either 0.001 or 0.01 learning rate, did not appear to work well for our model as the F1 score, precision and recall are fairly low because the models in these cases changed between predicting all negative and all positive values between folds. This is possibly underfitting the data and may be due to being stuck in a local minimum. Adam with a higher initial learning rate of 0.001 rather than 0.0001 appears more successful in optimizing the model. However, overall, the results are somewhat disappointing since even a powerful model such as LSTM with a few hidden layers is not able to achieve an accuracy significantly above 0.5 for the dataset. This is possibly because the dataset is difficult to build machine learning models around as historical stock data is very noisy and not filled with very noticeable patterns that models would be able to learn. This is especially true because the patterns appear to change significantly from year to year so predicting results for 5 years ahead is fairly difficult.

7 Discussions, Observations, Comparisons, Future Work

In terms of overall results, we found that SVM using SGD with an RBF kernel worked best with an 0.532 accuracy, 0.63 ROC AUC, 0.526 F1 score, 0.552 precision, and 0.752 recall. This is slightly better than the best performing logistic regression using bounded grid search with an overall 0.545 accuracy, 0.546 F1 score, 0.556 ROC AUC, 0.58 precision, and 0.655 recall. It is also slightly better than the best performing LSTM model which used Adam with an initial 0.001 learning rate which had 0.524 accuracy, 0.526 ROC AUC, 0.495 F1 score, 0.558 precision, and 0.5 recall.

Overall, it is possible that a model like SVM with RBF kernel which is fairly powerful but not as powerful as an LSTM model is best for our dataset because of how noisy stock data can be. It is possible that even with regularization through dropout for our LSTM, it is still overfitting the data and not performing as well as SVM. Logistic regression also performs slightly worse than SVM but slightly better than the LSTM model. However, the results for each model and optimization method are fairly similar and the differences are fairly small. It is possible that with other datasets or other train/validation splits that another model could perform the best by a slight margin as well. Overall, this indicates that although we tried various optimization methods, it is difficult to obtain a clear well-performing model compared to others on our dataset.

In terms of observations from our work, we found that optimizing various models such as logistic regression, SVM, and LSTM for our dataset was a difficult optimization problem. For each model, optimizers struggled to optimize the models successfully. This is possibly because the loss functions for each model are fairly smooth and there are many local minima that make it difficult for an optimizer to find the global minimum. More detailed discussion on each model/optimization method are included in the sections above.

In terms of our overall experimental design and approach, we found that testing various datasets like Apple, SPY, and AT&T yielded slightly different results due to the patterns found in each dataset. It is possible that some datasets would be easier to optimize machine learning models for. Also, in terms of problem formulation, binary classification appears easier to evaluate than predicting the daily price of a stock. This is because for some stocks, the daily movement in price is very small so it is difficult to determine how much a model is learning about the daily price changes. In comparison, for binary classification of price increase/decrease, if a model is able to accurately predict significantly more than half of days correctly and the dataset is balanced between increases/decreases then the model is likely learning useful patterns. In terms of feature engineering and selection, the implemented methods appeared to work moderately well. However, having more features such as public sentiment on a stock would also be interesting/useful to add to a model.

For future work, testing a larger variety of stocks/ETFs could be helpful to identify particular stocks/ETFs that we can successfully build accurate machine learning models for. It is possible that some or many stock/ETFs like the ones that we tested are difficult/infeasible to accurately predict future price changes for. Also, having accesss to more features on a stock could lead to better models and make them easier to optimize.

8 Contributions of Team Members

Lewis Wang worked on RNN/LSTM background research, implementation, theory, and results. Lewis also worked on finding the dataset, developing project goals, and contributed to designing the implementation details to ensure that they were consistent between our models. Lewis wrote the Abstract, RNN/LSTM, Discussions/Observations/Comparisons/Future Work sections of the final report.

Bryan Coronel took charge of the paper's introductory section. He conducted extensive research on various nonlinear optimization methods/theory such as basinhopping, particle swarm optimization, random search, and grid search, and explored ways to make them effective in iteratively refining a model. In general, he spearheaded the direction of the report from regression to classification task based on initial exploratory data analysis, eventually defining the target variable as price direction.

Alex Greene wrote the entire SVM section and coded/ran results of the section as well. While I did the SVM theory, I did not do the optimization method theory (which was Charles). I also wrote the implementation details section, which consists of the dataset, feature selection, and training. I developed the initial code for the project. I also spearheaded the group's general progress, first to complete work, report problems, set meeting times, etc.

Charles Troutman wrote the theory for each optimization method, including SMO, Coordinate Descent, Basin Hopping, PSO, ADMM, RNNs, SGD, and BPPT, for the three different models, SVM, logistic regression, and RNN/LSTM that we explored in our project. Charles conducted significant reading and learning of the different methods and combined the best sources to optimally provide a detailed report on the theory behind these optimization methods. In the report, Charles thoroughly explored and explained each optimization method to provide a comprehensive understanding for the reader.

9 References

1. Hochreiter, S. & Schmidhuber, J. LONG SHORT-TERM MEMORY. Neural Computation (1997). https://doi.org/10.1162/neco.1997.9.8.1735.

2. Correa-Álvarez, C.D., Salazar-Uribe, J.C. & Pericchi-Guerra, L.R. Bayesian multilevel logistic regression models: a case study applied to the results of two questionnaires administered to university students. Comput Stat (2022). https://doi.org/10.1007/s00180-022-01287-4.

3. Meng, Zeng, Li, Gang, Wang, Xuan, Sait, Sadiq & Yildiz, Ali. (2020). A Comparative Study of Metaheuristic Algorithms for Reliability-Based Design Optimization Problems. Archives of Computational Methods in Engineering. 28. 10.1007/s11831-020-09443-z.

4. Agnieszka Strzelecka, Agnieszka Kurdyś-Kujawska, Danuta Zawadzka, Application of logistic regression models to assess household financial decisions regarding debt, Procedia Computer Science, Volume 176, 2020, Pages 3418-3427, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2020.09.055.

5. Sherstinsky, A. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. Physica D: Nonlinear Phenomena (2020). https://doi.org/10.1016/j.physd.2019.132306.

6. Stitson, M. O., et al. "Theory of support vector machines." University of London 117.827 (1996): 188-191.

7. Lin, Chih-Jen. "Formulations of support vector machines: a note from an optimization point of view." Neural Computation 13.2 (2001): 307-317.

8. R Febrianti et al 2021 J. Phys.: Conf. Ser. 1725 012014

9. Vinkó, T., & Gelle, K. (2017). Basin Hopping Networks of continuous global optimization problems. Central European Journal of Operations Research, 25, 985-1006. https://doi.org/10.1007/s10100-017-0480-0.

10. El Dor, A., Clerc, M., & Siarry, P. (2012). A multi-swarm PSO using charged particles in a partitioned search space for continuous optimization. Computational Optimization and Applications, 53(1), 271-295. https://doi.org/10.1007/s10589-012-9492-6.

11. Wright, S. (n.d.). Coordinate Descent Algorithms. Retrieved April 29, 2023, from https://citeseerx.ist.psu.edu/viewdoc/downloaddoi=10.1.1.95.1418rep=rep1type=pdf.

12. Roberts, D., et al. (2022). The Principles of Deep Learning Theory: An Effective Theory Approach to Understanding Neural Networks. Cambridge University Press.

13. Kingma, D., & Ba, J. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. ICLR (2015). https://doi.org/10.48550/arXiv.1412.6980.

10 Appendix

10.1 SVM

10.1.1 Coordinate Descent

CV	Accuracy	ROC AUC Score	F1 Score	Precision	Recall	Confusion Matrix	Future Predictions
1	0.504	0.504	0.396	0.490	0.332	[[562, 279], [539, 268]]	[G, L, L, G, G]
2	0.522	0.533	0.463	0.530	0.411	[[522, 301], [486, 339]]	[G, G, G, G, G]
3	0.494	0.519	0.423	0.565	0.338	[[508, 236], [598, 306]]	[L, L, L, L, L]
4	0.461	0.500	0.449	0.547	0.381	[[397, 300], [589, 362]]	[G, G, G, G, G]
5	0.436	0.488	0.264	0.550	0.174	[[552, 137], [792, 167]]	[G, G, G, L, L]

10.1.2 SMO

For the RBF kernel:

CV	Accuracy	ROC AUC Score	F1 Score	Precision	Recall	Confusion Matrix	Future Predictions
1	0.490	0.475	0.657	0.490	1.0	[[0, 841], [0, 807]]	[G, G, G, G, G]
2	0.501	0.530	0.667	0.501	1.0	[[0, 823], [0, 825]]	[G, G, G, G, G]
3	0.549	0.515	0.709	0.549	1.0	[[0, 744], [0, 904]]	[G, G, G, G, G]
4	0.577	0.454	0.732	0.577	1.0	[[0, 697], [0, 951]]	[G, G, G, G, G]
5	0.582	0.511	0.736	0.582	1.0	[[0, 689], [0, 959]]	[G, G, G, G, G]

For the linear kernel:

CV	Accuracy	ROC AUC Score	F1 Score	Precision	Recall	Confusion Matrix	Future Predictions
1	0.490	0.477	0.657	0.490	1.0	[[0, 841], [0, 807]]	[G, G, G, G, G]
2	0.501	0.463	0.667	0.501	1.0	[[0, 823], [0, 825]]	[G, G, G, G, G]
3	0.549	0.533	0.709	0.549	1.0	[[0, 744], [0, 904]]	[G, G, G, G, G]
4	0.577	0.524	0.732	0.577	1.0	[[0, 697], [0, 951]]	[G, G, G, G, G]
5	0.582	0.492	0.736	0.582	1.0	[[0, 689], [0, 959]]	[G, G, G, G, G]

10.1.3 SGD

For the RBF kernel:

CV	Accuracy	ROC AUC Score	F1 Score	Precision	Recall	Confusion Matrix	Future Predictions
1	0.516	0.526	0.555	0.505	0.616	[[354, 487], [310, 497]]	[G, G, G, G, G]
2	0.501	0.514	0.654	0.501	0.943	[[47, 776], [47, 778]]	[G, G, G, G, G]
3	0.546	0.524	0.667	0.558	0.830	[[150, 594], [154, 750]]	[L, L, L, L, L]
4	0.557	0.551	0.625	0.611	0.639	[[310, 387], [343, 608]]	[G, G, G, G, G]
5	0.541	0.514	0.650	0.584	0.734	[[187, 502], [225, 704]]	[L, L, L, L, L]

For the Linear kernel:

CV	Accuracy	ROC AUC Score	F1 Score	Precision	Recall	Confusion Matrix	Future Predictions
1	0.510	0.520	0.515	0.500	0.532	[[412, 429], [378, 429]]	[G, G, G, G, G]
2	0.527	0.524	0.552	0.525	0.583	[[387, 436], [344, 481]]	[G, G, G, G, G]
3	0.530	0.526	0.595	0.564	0.629	[[305, 439], [335, 569]]	[L, L, L, L, G]
4	0.558	0.535	0.648	0.600	0.705	[[250, 447], [281, 670]]	[G, G, G, G, G]
5	0.534	0.517	0.638	0.582	0.707	[[202, 487], [281, 678]]	[L, L, L, L, L]

10.2 RNN/LSTM Cross-validation Tables

For ADAM (0.001 initial learning rate (lr)):

CV	Accuracy	ROC AUC	F1 Score	Precision	Recall	Confusion Matrix
1	0.513	0.540	0.168	0.519	0.100	[[764, 75], [727, 81]]
2	0.528	0.545	0.545	0.525	0.566	[[403, 421], [357, 486]]
3	0.505	0.472	0.569	0.545	0.595	[[295, 449], [366, 537]]
4	0.520	0.557	0.521	0.614	0.453	[[425, 271], [520, 431]]
5	0.556	0.515	0.673	0.589	0.787	[[161, 527], [205, 754]]

For ADAM (0.0001 initial learning rate (lr)):

CV	Accuracy	ROC AUC	F1 Score	Precision	Recall	Confusion Matrix
1	0.538	0.540	0.424	0.546	0.347	[[606, 233], [528, 280]]
2	0.522	0.531	0.464	0.528	0.414	[[519, 305], [482, 341]]
3	0.488	0.475	0.530	0.534	0.526	[[329, 415], [428, 475]]
4	0.576	0.549	0.698	0.592	0.850	[[140, 556], [143, 808]]
5	0.464	0.496	0.342	0.599	0.239	[[535, 153], [730, 229]]

For SGD (0.001 learning rate (lr)):

CV	Accuracy	ROC AUC	F1 Score	Precision	Recall	Confusion Matrix
1	0.509	0.492	0.000	0.000	0.000	[[839, 000], [808, 000]]
2	0.500	0.546	0.000	0.000	0.000	[[824, 000], [823, 000]]
3	0.452	0.462	0.000	0.000	0.000	[[744, 000], [903, 000]]
4	0.577	0.540	0.732	0.577	1.000	[[000, 696], [000, 951]]
5	0.582	0.480	0.736	0.582	1.000	[[000, 688], [000, 959]]

For SGD (0.01 learning rate (lr)):

CV	Accuracy	ROC AUC	F1 Score	Precision	Recall	Confusion Matrix
1	0.509	0.532	0.000	0.000	0.000	[[839, 000], [808, 000]]
2	0.500	0.517	0.000	0.000	0.000	[[824, 000], [823, 000]]
3	0.452	0.457	0.000	0.000	0.000	[[744, 000], [903, 000]]
4	0.577	0.543	0.732	0.577	1.000	[[000, 696], [000, 951]]
5	0.582	0.507	0.736	0.582	1.000	[[000, 688], [000, 959]]