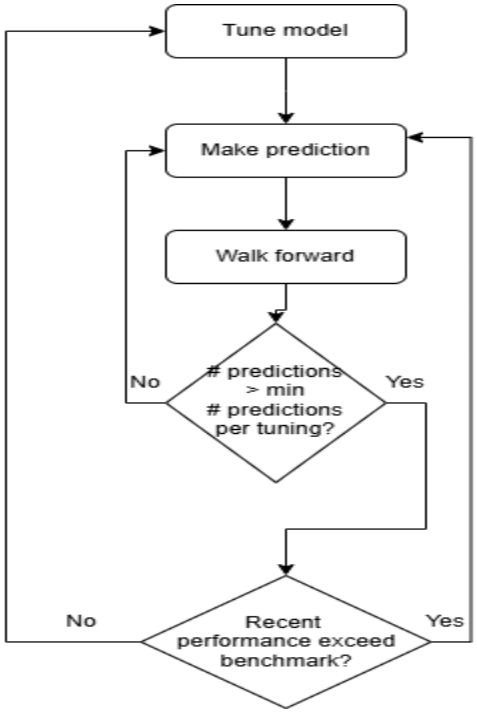
Overview

This repository contains code for the basis of a stock prediction algorithm. It utilizes various machine learning principles and architectures to accomplish this endeavor. This is not the final version of the project; some ideas remain proprietary, while others are yet to be implemented due to capital and time constraints. Furthermore, some implemented concepts are omitted for brevity, as their impact on the algorithm or its understanding does not justify the complexity required to explain them.

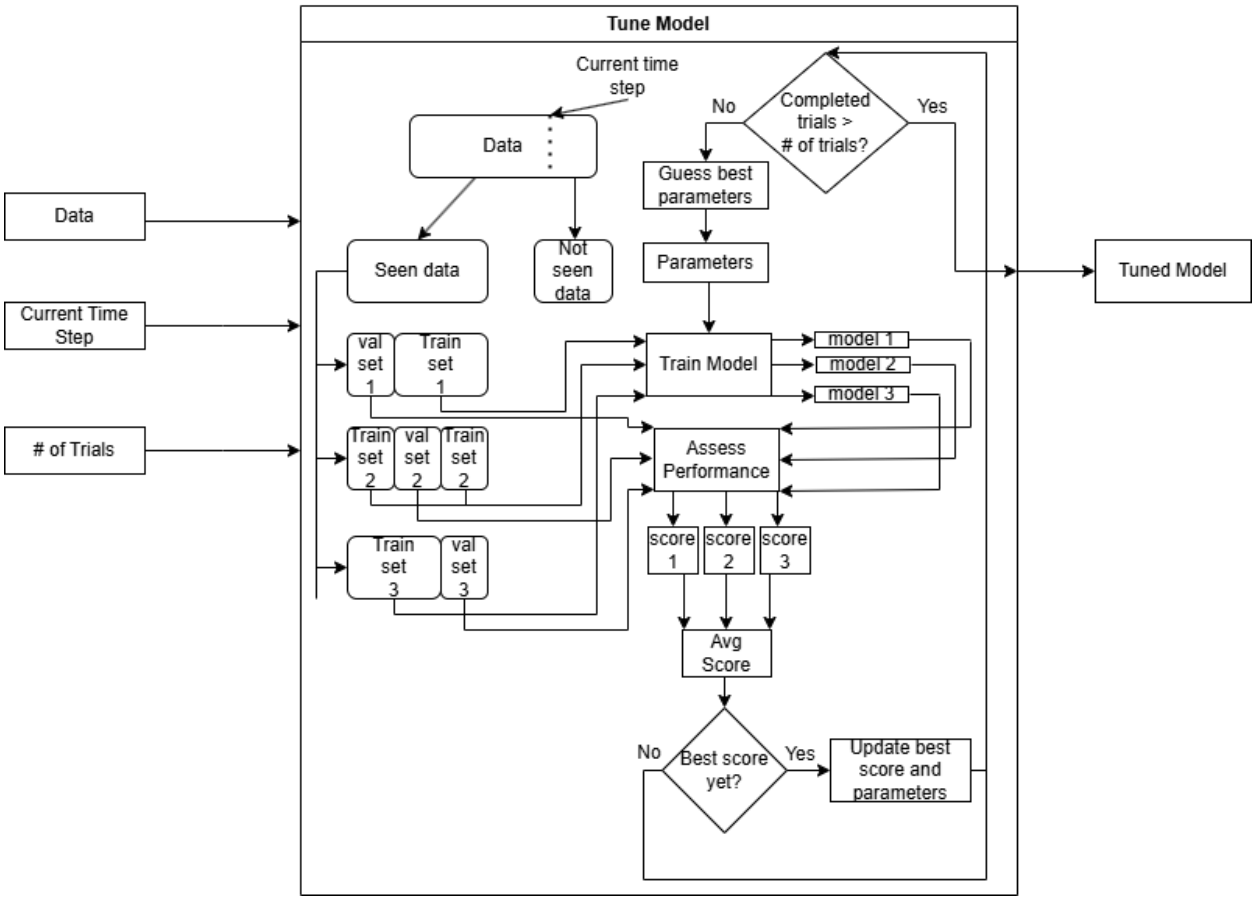
Method of Evaluation

To ensure our evaluation is meaningful we have to evaluate it with "out of sample" data. This means that the model cannot be trained on the data in which we assess its accuracy. This would be analogous to letting a student study the answer key and thus testing would not be meaningful. In order to do so we employ a concept called walk forward validation. In walk forward validation we have the model make predictions and then we walk it forward through time. Some forms of validation do not walk forward but rather are discrete immutable sets; however, this may be inadequate in this domain due to the potential predictive power of recent data. For example, patterns prior to 2008 may not be holding after, and may require that the model is trained with the post-recession data.



Nested Cross Validation

So not only do we need the ability to test it out of sample but we need the ability to tune certain values relative to our domain. For example, when fitting a linear regression model, we may want to include data from more than one time step ago for predicting; ie, calibrate how many days back we want to look. So within our "seen data", the data we have walked forward through, we want to tune these parameters. The solution is to let our data take turns being part of the validation set and averaging over these conditions. This process we will refer to as tuning the model.



Employed Architectures

As seen in the tune model diagram we are not restricting ourselves to any one type of model. We can generically apply this process to almost any type of model that can be trained on data and has hyperparameters to tune. This is intentional because different models can capture different types of patterns and in a domain as complex as stock prediction one may not be enough. Thus, while there are many more to employ, we are currently using XGBoost, linear regression, decomposition linear regression, recurrent neural networks, and a transformer with convolutional and recurrent neural network layers. These were selected because ideally they will provide different predictive insights and they cost different amounts to run.

XGBoost is a gradient boosted decision tree model; which, effectively means that it tries to construct flow charts to make decisions and it uses calculus to figure out how to do so. For example, it may say if the moving average of the stock price is > x, and its volatility < y, then predict z. It uses calculus to figure out those values for x, y, z as well as which features to consider. Linear regression involves weighting features, in combination with a bias term, in order to predict an output. For example, maybe we determine that the best modeling of the stock price is: stock price = (1.05 * yesterday's price) - (.95 * two day's ago price) + 5.13. Decomposition linear regression is an extension of linear regression where it segments itself. For example, it may say stock price = linear regression of long term trend + linear regression of local patterns. A more intuitive example may be modeling temperature through the year and having one model the average temperature per month and one modeling the nuances of the current month (maybe it is a relatively hot december for example).

The remaining architectures are all neural network based which makes them incredibly difficult to explain so much so that they are commonly referred to as "black boxes", due to the lack of transparency within itself. With that said, some analogies can be employed to get a rough understanding of their processes and the first of them being that they are neural networks. We call them neural networks because we like to think of them analogous to our neurons where they learn how to connect themselves and share information amongst each other. Recurrent neural networks get their name because they have an added temporal component that allows them to understand previous input. This allows it to capture nuances like the differences between "only she loves him" and "she loves only him". Convolutional neural networks learn spatial relationships like if one is searching an environment for dams that they may want to be looking at the areas with water. We can utilize this concept by having the spatial component be time rather than say proximity. Lastly, transformers are a type of neural network that learns what part of the data that it should be paying attention to. For example, we can think of this difference as a student that understands to study only his biology notes for his upcoming biology test rather than one that would study all of his subject’s notes.

Data Construction And Selection

The most important aspect of any machine learning pipeline is the data in which it operates on. This does not just refer to the quantity of data, or even the quality of the data, but knowing what to include, what to modify, and what to create. In our machine learning pipeline we utilize these concepts to get as much as we can out of the data as possible. The most simple of these operations are technical indicators. Our data starts with simple financial features related to each stock like its close, open, volume, high, low. Out of these, we create new features like its percent change of open to close, moving averages that percent change, or the tails of the bell curve of those values. We can do this not just for our target stock but also ones that we believe are closely related like maybe Microsoft and Apple which we can determine through a correlation analysis. Furthermore, we can make up stocks that are combinations of others and test how predictive they are. For example, say our target stock is Microsoft, maybe instead of Apple we determine that ((2 * Apple) - (3 * Nvidia)) is even more predictive for Microsoft. We test how predictive it is through cointegration analysis which seeks to see if two variables travel through time together in a predictable manner. For example, maybe when gas prices go up, Amazon's stock prices go down because they are sensitive to it as a resource.

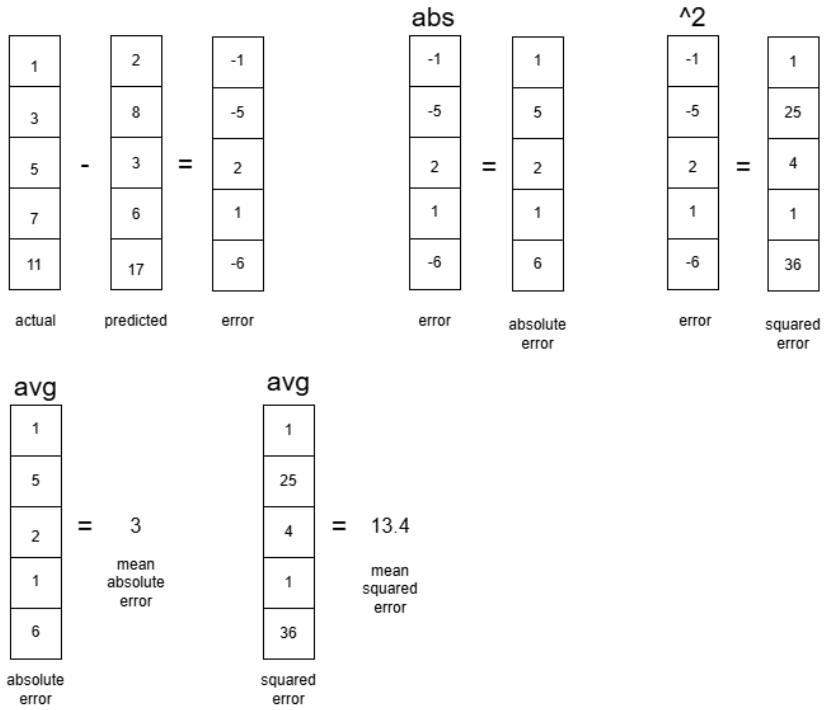
An issue can arise from our feature creation though, what if we have too many features? Imagine you are trying to manage a team task where you have to assign every single member a role on how they can help. An additional helper may be great, but having to assign tasks to 10,000 additional helpers may be detrimental. So it becomes clear that we need to optimize the number of helpers, or in this case, the number of features. This is actually relatively not hard to do and is done within the model tuning process which was previously shown where we try different values and find the optimal value using calculus.

However, a much harder problem is determining which features to keep and which to let go. Imagine the task at hand was creating the best trivia team possible. If there are too many people then it may be difficult to figure out who to listen to. And if the team were constructed of the best N players then there is risk of them being experts in the same subjects and not providing new value. Thus one would want to construct the team such that they minimize shared knowledge and maximized the knowledge of the potential topics. So in our domain, we want to maximize the predictive power (relevance) and minimize the correlation amongst each other (redundancy), this algorithm is known as MRMR or maximum relevance minimum redundancy. It selects features incrementally such that each feature maximizes the value of correlation with target minus the average correlation with selected features. A slight tweak can be made to this algorithm in which we instead maximize the value of correlation with target minus the correlation of the feature it is most correlated with. I have not found this algorithm elsewhere but I am not going to claim it as my invention; however, for now, it needs a name and I am calling it greedy feature selection. Thus we have these two algorithms rank the features in order it would select them, and then we have the tuning process select which algorithm to use, and how many features to use.

Loss Function Selection

Loss functions are a very important aspect of any machine learning algorithm. Their purpose is to inform the algorithm how well it is performing. Now one may think that there should be just one universal one but different domains have different constraints and thus benefit from different methods of analyzing. For example, imagine one were constructing a model for diagnosing a very deadly contagious disease that requires quarantining. The punishment for incorrectly telling someone they have it when they don't is that they have to be quarantined; whereas the punishment for incorrectly telling them they are healthy is the spreading of a deadly disease and subsequently death. Thus, we may have different tolerances to these different types of errors and we have to decide how to weigh them.

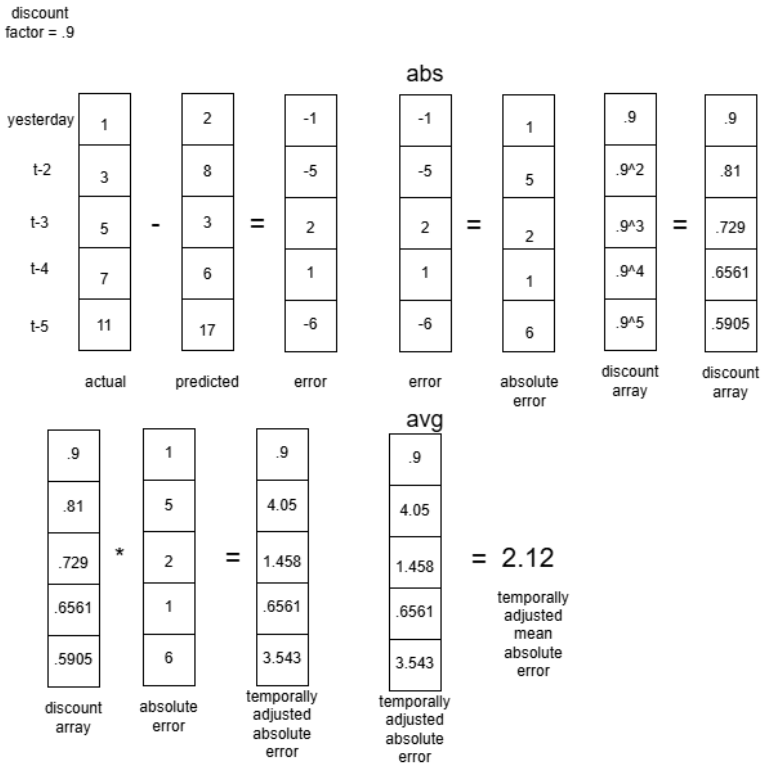
In the current implementation we only deal with regressive models which means we get a numeric output rather than a classification. So we will focus on two of the employed loss functions to illustrate the importance of correct selection and modification. The first loss function is called mean absolute error. In this we subtract the predicted values - the actual values = the errors and then we find the average error value. This is one of the most popular loss functions due to its high interpretability and alignment with the problem at hand; however, it may not account adequately for large outliers. The other one is mean squared error and it is represented by the (predicted values - the actual values)^2 = the errors and then we find the average value. This is another very popular loss function and is often selected over mean absolute error because it punishes larger errors more heavily. Below I have illustrated these concepts.



Loss Function Modification

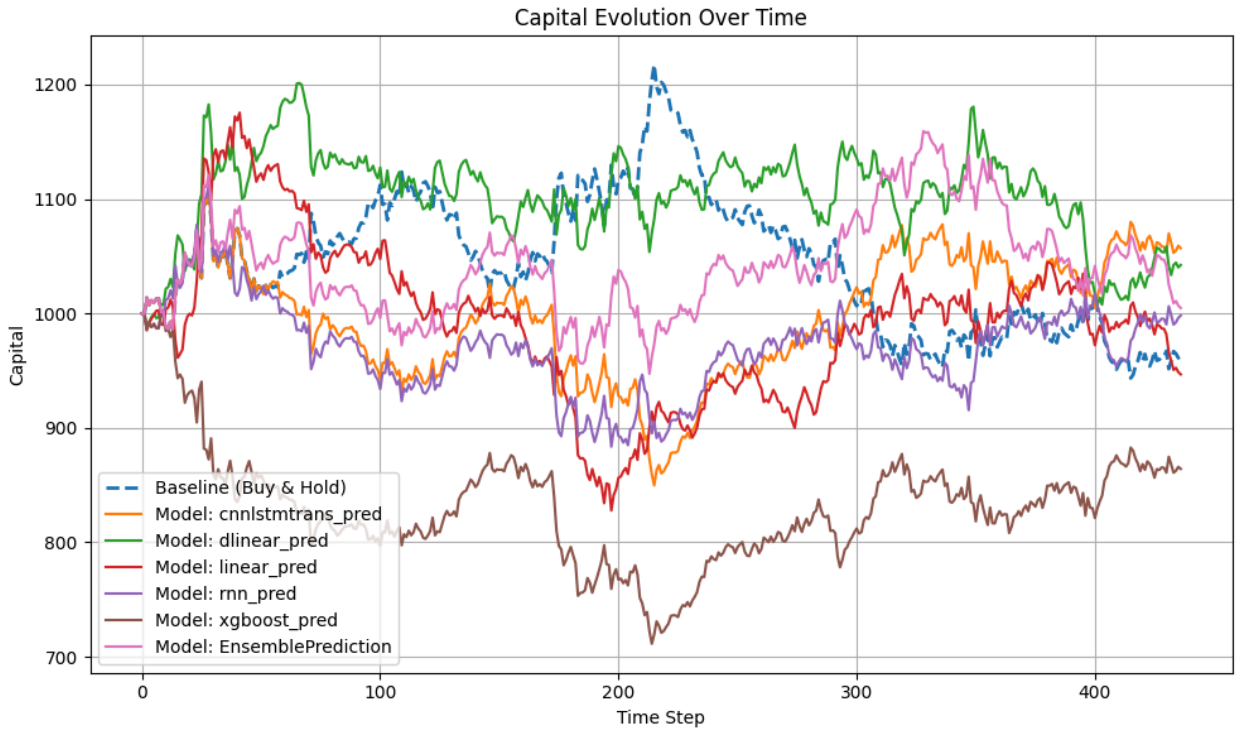
In some domains selection of loss function may not be enough and the stock domain is a particularly compelling one for this case. For example, imagine that we predict the stock price of XYZ tomorrow will go up 1%. If tomorrow the stock goes to 1% we will report the optimal value to our error function (0). However, if it goes up 2% or down 1%, we will report an error of 1% but in one case we will profit and in the other we will lose money.Thus, it doesn't make sense to punish these errors equally because that is not quite aligned with our objective. So, we can introduce a parameter to our loss functions so that we punish overprediction and underprediction differently and we can have this parameter selected in the tuning process.

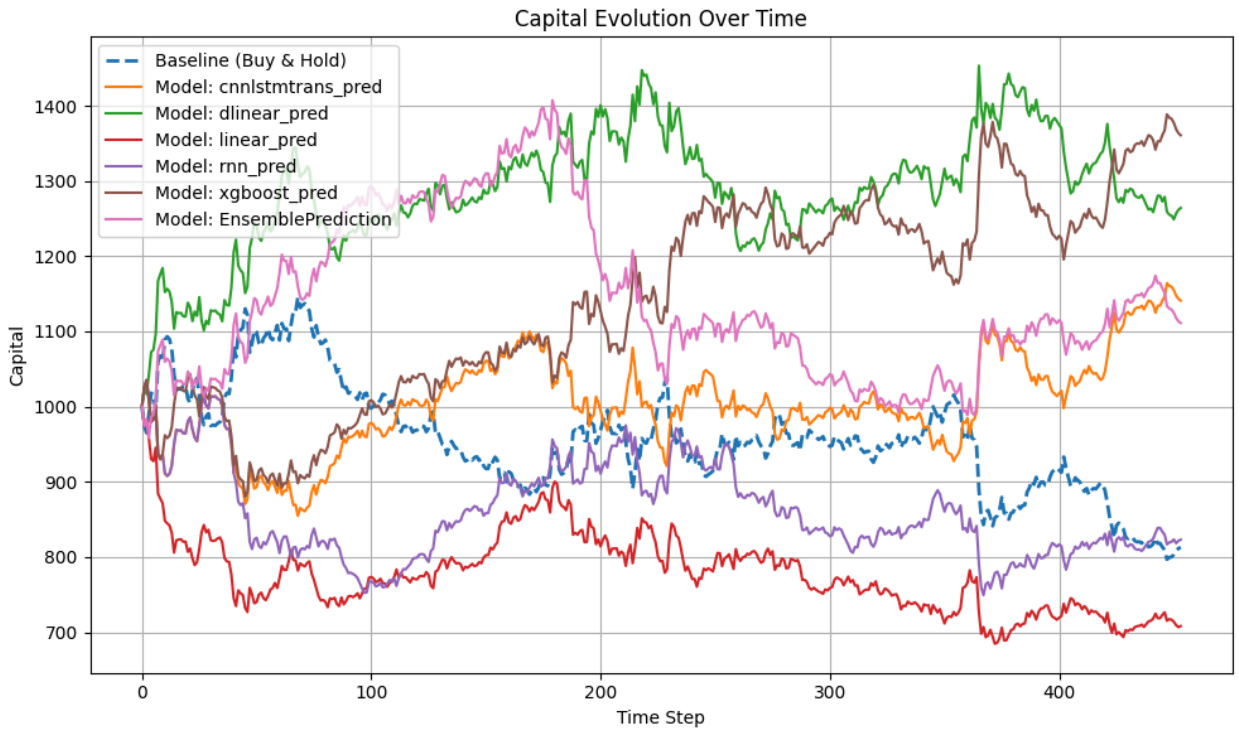
The stock prediction domain also suffers from another challenge that makes it a prime candidate for loss function modification, non-stationarity. Non-stationarity refers to the statistical principle that certain statistical properties like the mean, variance, etc. evolve overtime. One of the best examples of this is climate change because while there are interannual and intraseasonal fluctuations, there is a fundamental underlying change happening to the statistical properties. Thus, averaging the error over historical data without consideration to the recency of the data can lead to overly optimistic predictions. Similarly, we may find it hard to predict Amazon's stock price if we give serious weight to when it was a bookstore. So, we can introduce another parameter to our loss functions called a discount factor. The discount factor is selected during the tuning process such that it is between 0 and 1. The discount factor, we take to the nth power where n is the number of time steps from the present day, and multiply our error by it. I have modified the previous error implementation to illustrate this concept.

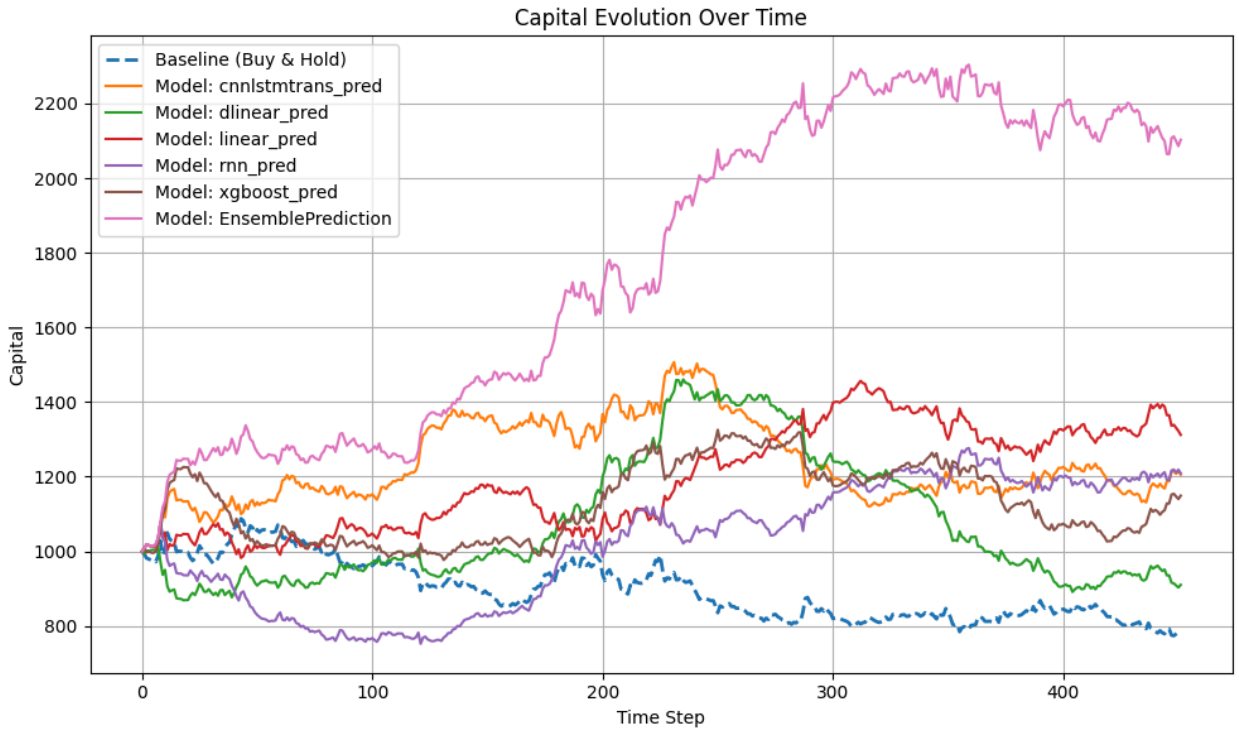


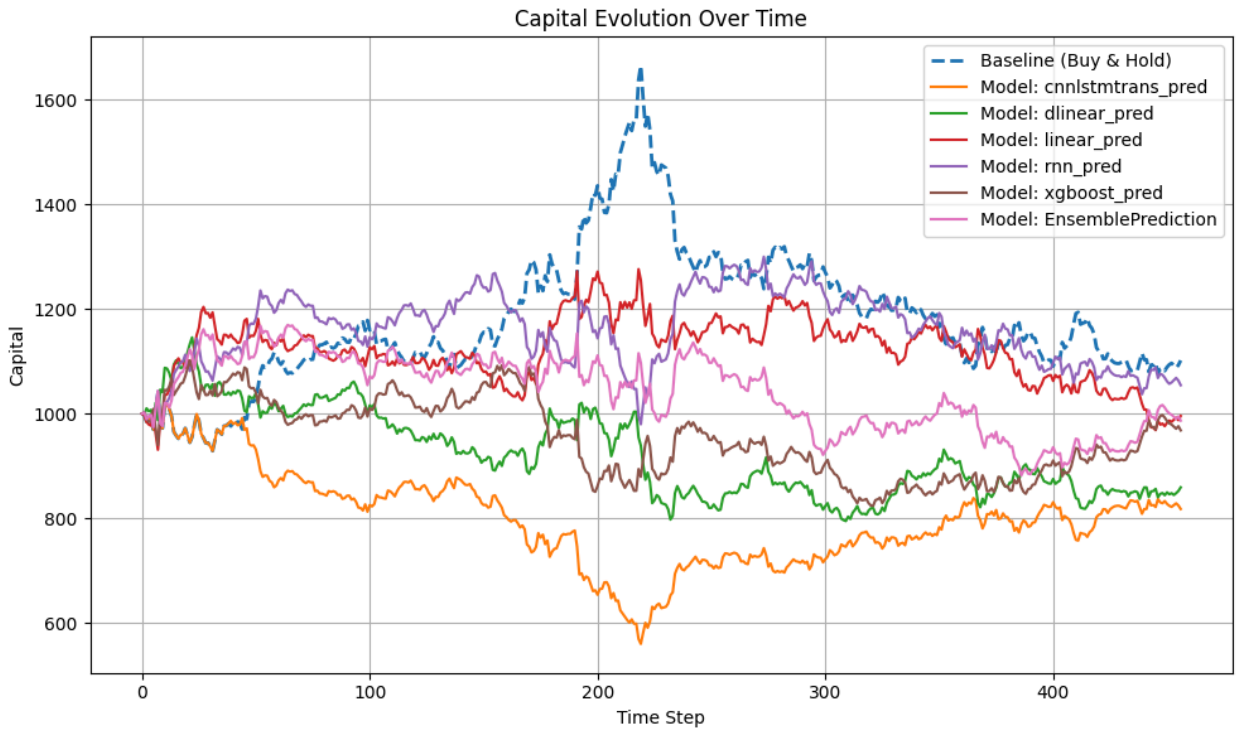
Results

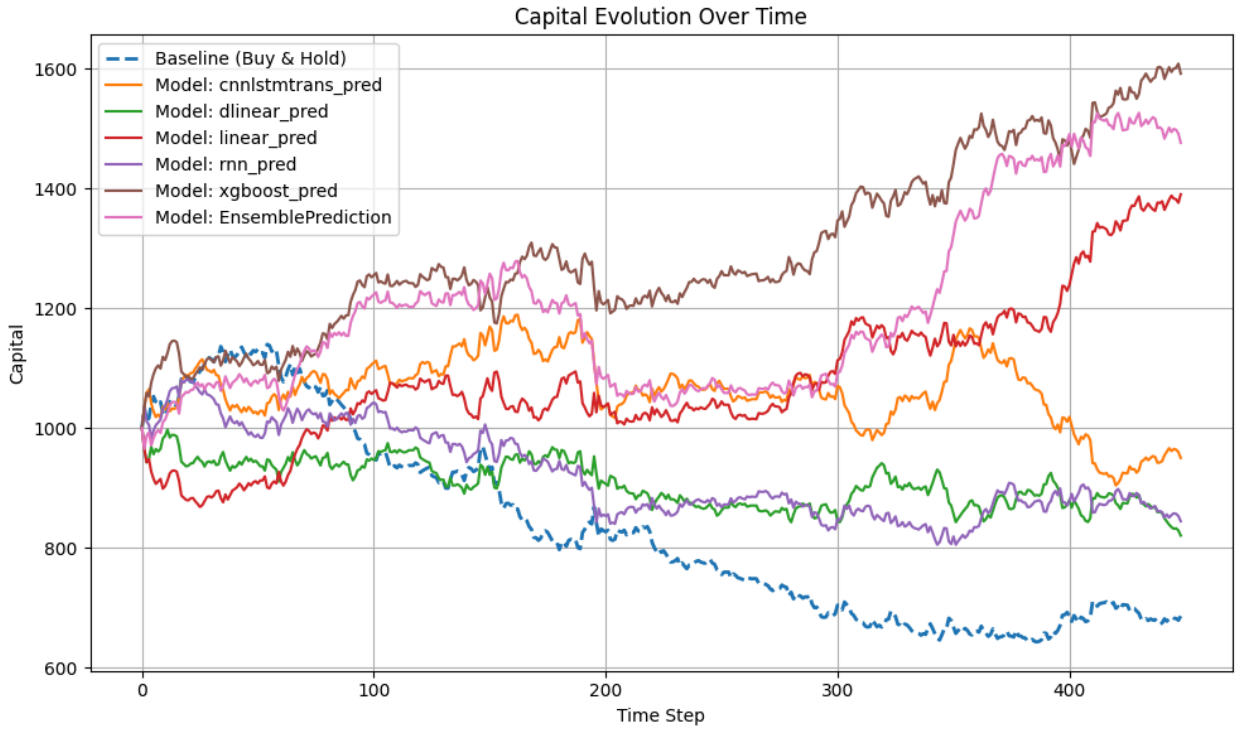
To evaluate this algorithm, we tested the five model types on six different stocks. We then trained an ensemble model to interpret their predictions and generate a final output. Given the limited data, we opted for simplicity: the model tracks past performance over a set number of steps and selects the best-performing model's prediction. This look-back parameter was tuned using nested cross-validation and walk-forward validation. This strategy resulted in beating the market by 56% and an average profit of 33%. Posted below are the charts of the final evaluation for ALL, AMGN, BAX, GS, PG, and VZ respectfully.

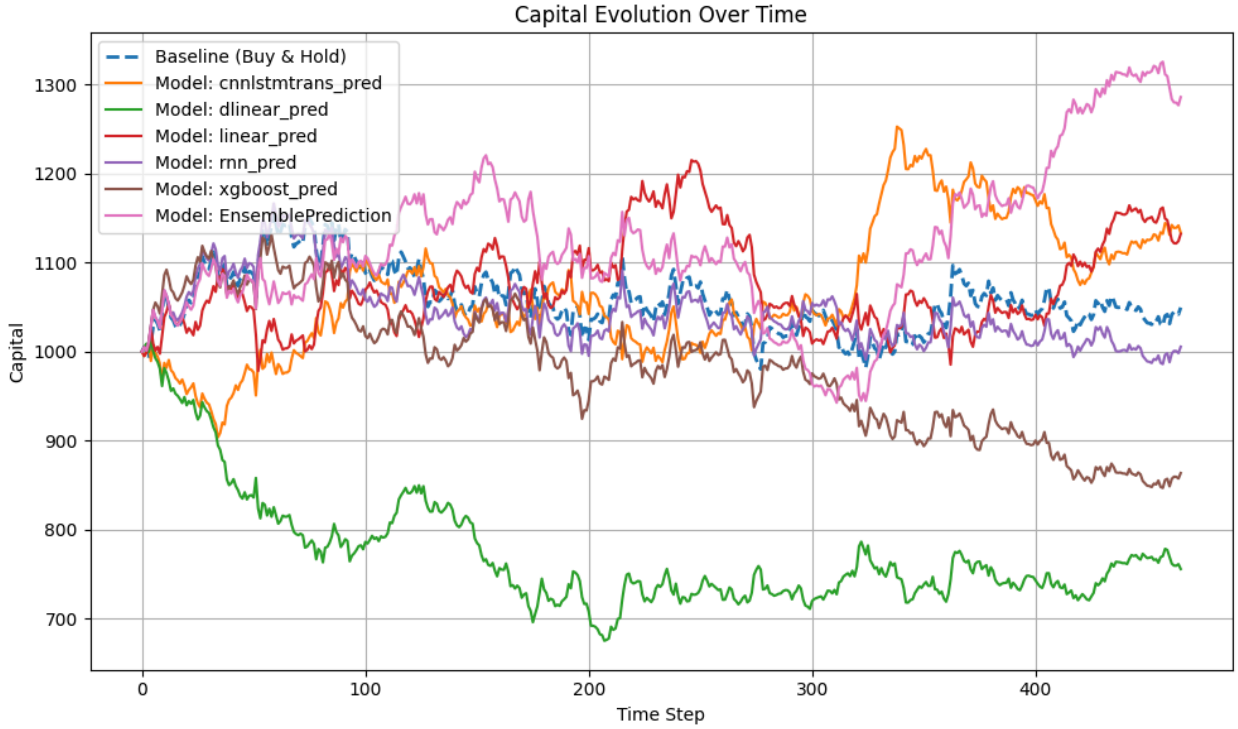



Capital Evolution Over Time
Baseline (Buy & Hold)
Model: cnnlstmtrans_pred
Model: dlinear_pred
Model: linear_pred
Model: rnn_pred
Model: xgboost_pred
Model: EnsemblePrediction
Capital
Time Step
700
800
900
1000
1100
1200
1300
1400
0
100
200
300
400


Capital Evolution Over Time
Baseline (Buy & Hold)
Model: cnnlstmtrans_pred
Model: dlinear_pred
Model: linear_pred
Model: rnn_pred
Model: xgboost_pred
Model: EnsemblePrediction
Capital
Time Step
800
1000
1200
1400
1600
1800
2000
2200
0
100
200
300
400


Capital Evolution Over Time
Baseline (Buy & Hold)
Model: cnnlstmtrans_pred
Model: dlinear_pred
Model: linear_pred
Model: rnn_pred
Model: xgboost_pred
Model: EnsemblePrediction
Capital
Time Step
1600
1400
1200
1000
800
600
0
100
200
300
400


Capital Evolution Over Time
Baseline (Buy & Hold)
Model: cnnlstmtrans_pred
Model: dlinear_pred
Model: linear_pred
Model: rnn_pred
Model: xgboost_pred
Model: EnsemblePrediction
Capital
Time Step
1600
1400
1200
1000
800
600
0
100
200
300
400


Capital Evolution Over Time
Baseline (Buy & Hold)
Model: cnnlstmtrans_pred
Model: dlinear_pred
Model: linear_pred
Model: rnn_pred
Model: xgboost_pred
Model: EnsemblePrediction
Capital
Time Step
1300
1200
1100
1000
900
800
700
0
100
200
300
400