Stock Return Prediction and Optimal Feature Selection Using Machine Learning

Group Member's Names	Student ID	Approximate Contribution
Xue		16.67%
Zheng		16.67%
Wang		16.67%
Lu		16.67%
Yang		16.67%
Liang		16.67%

December 30, 2024

Abstract

To predict stock returns, we compare a range of machine learning (ML) methods with traditional econometric approaches. Employing linear regression (including ElasticNet), tree-based models (Random Forests, Gradient Boosting Trees, LightGBM), and neural networks, we perform a comparative analysis of predictive performance. Results indicate that nonlinear ML models (excluding ElasticNet) achieve superior predictive accuracy compared to linear methods. Furthermore, we identify key predictive factors contributing to the economic interpretability of the models, finding that price trends (momentum), liquidity, and volatility are consistently among the most influential features.

Contents

1 Introduction

The application of machine learning (ML) in financial prediction, particularly in stock return forecasting, has gained increasing importance. Traditional econometric models have struggled to account for the complexity of financial data, which often involves high-dimensional predictors, nonlinear relationships, multicollinearity, and time-varying correlations. These limitations have driven the shift towards ML methods, which excel in capturing intricate patterns and handling large, complex datasets, leading to improved forecasting accuracy in financial markets.

Ensemble methods, such as Random Forests, Gradient Boosting Trees, and LightGBM, as well as Neural Networks, have demonstrated remarkable success in predicting stock returns and constructing portfolios with strong risk-adjusted performance. These methods are well-suited to handle the nonlinearities and feature interactions inherent in financial data. However, despite their predictive power, ML models face challenges such as limited interpretability compared to traditional models and high computational demands, as well as the risk of overfitting due to noisy data. This has prompted ongoing research into improving model transparency and integrating the strengths of both ML and traditional econometric approaches, fostering more robust, balanced, and interpretable solutions for financial forecasting.

Significant advancements in ML for financial forecasting include techniques like regularized linear models (ElasticNet), dimension reduction methods (PCR and PLS), ensemble methods, and neural networks. These methods not only offer better predictive performance but also provide valuable insights into the underlying drivers of asset prices. The increasing availability of high-frequency and unstructured financial data further underscores the need for advanced predictive models. By combining ML with traditional financial theory, new opportunities arise to address long-standing challenges in asset pricing, improve portfolio construction, and provide deeper insights into the fundamental causes of asset price movements. As this field continues to evolve, it holds the potential to reshape both academic research and practical applications in finance, driving innovation across diverse areas of financial analysis and unlocking new pathways for decision-making in increasingly complex financial markets.

2 Literature Review

Stock return prediction has long been an important research topic in the field of finance, attracting considerable attention from scholars. With the rapid growth of financial market data, particularly high-dimensional and complex market feature data, traditional linear asset pricing models have become increasingly inadequate. In this context, machine learning (ML) techniques, with their ability to handle nonlinear relationships, recognize complex patterns, and adapt to large-scale data, have gradually become the core tool for stock return prediction. An increasing number of researchers are turning to nonlinear machine learning models to uncover the underlying complex relationships in financial markets and enhance prediction accuracy.

Gu et al. (2020) proposed an innovative framework that utilizes machine learning models to predict stock returns, comparing them with traditional linear models. Their research indicates that machine learning methods, especially nonlinear models, can effectively capture complex nonlinear patterns in stock returns, significantly improving prediction accuracy. Compared to traditional regression analysis methods, machine learning models demonstrate clear advantages in handling the complexity and diversity of financial data. For example, Support Vector Machine (SVM) in financial data Kim (2003) proves that machine learning models are more effective in dealing with high-dimensional data. Additionally, deep learning models have received considerable attention for stock return prediction. Fischer & Krauss (2018) showed that deep neural networks can capture complex patterns that traditional methods fail to identify, improving prediction accuracy.

Data quality and processing are fundamental to the application of machine learning in the financial domain. Gu et al. (2020) ensured the quality of historical trading data through efficient data preprocessing and standardization, successfully applying it to predict stock risk premiums. Data cleaning and standardization help eliminate noise and outliers, thus improving model accuracy and reliability. Furthermore, Gu et al. employed high-dimensional data processing techniques, identifying the most relevant features from a large number of financial variables, which enhanced the model's predictive power. This technique aligns with feature selection and dimensionality reduction methods. For example, Princi-

pal Component Analysis (PCA) (Jolliffe, 2002) is widely used in financial data processing to reduce dimensionality while retaining key features, thus improving the effectiveness of machine learning models. Similarly, LASSO (Tibshirani, 1996) and Elastic Net (Zou & Hastie, 2005) feature selection methods are able to identify the most predictive variables from high-dimensional data, enhancing model accuracy.

In our study, we also adopted similar data preprocessing steps, including rigorous data cleaning and standardization, ensuring high-quality datasets, and laying a solid foundation for subsequent machine learning model training.

In this study, we not only replicated the nonlinear models used by Gu et al. (2020) but also introduced more modern machine learning techniques, such as Gradient Boosting and Random Forest. These ensemble learning models are better suited to capture complex patterns in the data, especially in high-dimensional and multivariate scenarios, demonstrating excellent predictive capabilities. Gradient Boosting Trees (GBT) (Chen & Guestrin, 2016) improve prediction accuracy by progressively weighting previous predictions. Meanwhile, Random Forest (Breiman, 2001) enhances model stability and accuracy by constructing multiple decision trees and using a voting mechanism to reduce overfitting.

In our research, these models were applied to stock return prediction and compared with other machine learning methods (such as SVM and traditional regression models) to further validate their superiority in financial data applications.

Beyond pure predictive accuracy, we also pay particular attention to the economic significance of the predictive factors. By incorporating macroeconomic factors and market behavior data, we gain a better understanding of the real-world impact of these factors. For example, certain financial indicators may play a more prominent role during specific market phases, while other factors may exhibit varying performance across different economic cycles. By integrating these non-financial factors into our model, we ensure that machine learning models not only provide high predictive accuracy but also offer predictions of economic relevance, providing valuable insights for investors and decision-makers. Additionally, combining volatility forecasting (Engle, 2001); Bollerslev, 1986 with machine learning models enables dynamic adjustments based on market conditions, resulting in more precise predictions.

Our study also compares machine learning methods with traditional financial models, clearly highlighting the advantages of machine learning in stock return prediction. Existing studies indicate that machine learning models outperform traditional models in predictive power, especially in modeling complex nonlinear relationships (He & Liao, 2021). For instance, regression analysis (Timmermann, 2018) struggles to capture the complex nonlinear features of financial markets, while machine learning models, particularly ensemble learning and deep learning, can model these complex relationships more effectively, improving prediction accuracy.

Overall, our research expands upon the machine learning framework proposed by Gu et al. (2020) in several ways, significantly improving both the accuracy and interpretability of stock return predictions. By employing efficient data preprocessing and feature selection techniques (such as LASSO and PCA), we ensured high-quality data input and effectively identified the most relevant features in high-dimensional data, boosting the model's predictive power. Furthermore, by incorporating Gradient Boosting and Random Forest ensemble learning methods, we captured complex patterns in the data, reduced overfitting, and enhanced model stability and adaptability. Through a comparative analysis of different machine learning models, we have explored the potential of machine learning in stock return prediction, offering new perspectives and methods for future research.

3 Methodology

3.1 Sample Splitting

We partition the dataset based on the method proposed by Gu, S., Kelly, B., & Xiu, D. (2020), dividing the 64 years of data into three subsets: 18 years for the training set (1957–1974), 12 years for the validation set (1975–1986), and 34 years for out-of-sample testing (1987–2020). This time-series approach ensures the preservation of the temporal order of data, which is crucial for avoiding data leakage and overfitting.

To further enhance the model's robustness, we adopt a hybrid partitioning strategy that integrates aspects of the fixed, rolling, and recursive split methods, with a particular emphasis on the rolling window approach. This allows the model to continu-

ously adapt to the most recent data, offering a dynamic view of market trends while avoiding the static nature of fixed partitions.

Under the rolling window method (Figure 1), the training set is progressively expanded each year by adding one additional year of data, while the validation and test sets remain fixed in length throughout the study. This preserves the temporal structure of the data and ensures that the model is consistently evaluated on the same validation set, reducing the risk of bias. Moreover, by avoiding cross-validation, we prevent future data leakage, making this methodology especially appropriate for time-series forecasting.

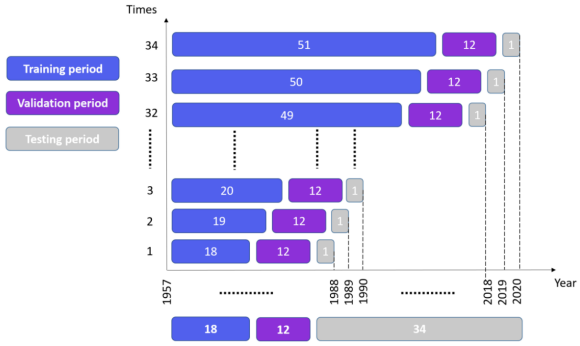


Figure 1: Rolling Window Approach for Temporal Data Partitioning

3.2 Linear Regression

Accurate prediction of asset returns is one of the most critical challenges in financial research and quantitative modeling. To systematically study the efficacy of various regression models for this purpose, we define a unified theoretical framework for asset return prediction. This framework models the process of return generation as follows:

$$r_{i,t+1} = f(\mathbf{x}_{i,t}) + \epsilon_{i,t+1}$$

In this model, $r_{i,t+1}$ represents the return of firm i at time $t+1$, which serves as the dependent variable. These returns could take various forms, such as monthly stock returns, dividends, or other types of returns. The vector $\mathbf{x}_{i,t}$ denotes the set of predictor variables observed at time t for firm i. These predictors may include firm-specific characteristics like size or the book-to-market ratio, as well as macroeconomic factors such as interest rates and inflation, or market variables like momentum and volatility.

The function $f(\cdot)$ is a predictive function that maps the predictor variables $\mathbf{x}_{i,t}$ to the expected return at time $t+1$. This function is model-specific and is estimated in different ways depending on the regression technique applied. Lastly, $\epsilon_{i,t+1}$ is the error term, representing all the unexplained variability in $r_{i,t+1}$. It is assumed to be independently and identically distributed (i.i.d.) with a mean of zero and finite variance.

The primary goal of this study is to identify the most effective predictive function $f(\cdot)$ by comparing the predictive performance of various regression models. We are particularly concerned with the models' generalization capabilities, which we evaluate using their performance on out-of-sample datasets. The evaluation metric is the R^2 value, which measures the proportion of variance in the dependent variable that the model can explain. This unified framework ensures a consistent baseline for comparing models, with each model imposing different assumptions on $f(\cdot)$ and the structure of the data.

3.2.1 Simple Linear Regression

Simple Linear Regression (SLR) assumes that the relationship between the predictor variables $\mathbf{x}_{i,t}$ and the response variable $r_{i,t+1}$ is linear. The functional form of the model is:

$$r_{i,t+1} = \beta_0 + \mathbf{x}_{i,t}^T \boldsymbol{\beta} + \epsilon_{i,t+1}$$

In this model, β_0 represents the intercept term, which corresponds to the baseline level of returns when all predictors are equal to zero. The vector $\boldsymbol{\beta}$ consists of the regression coefficients, where each coefficient β_j quantifies the effect of the j-th predictor on the response variable. The vector $\mathbf{x}_{i,t}$ denotes the set of predictor variables for firm i at time t, which may include numerical features derived from firm-specific or macroeconomic data. Finally, $\epsilon_{i,t+1}$ is the error term, which captures the deviations of the observed returns from the model's predictions. It is assumed to be normally distributed with a mean of zero.

The coefficients β_0 and $\boldsymbol{\beta}$ are estimated using the method of Ordinary Least Squares (OLS), which minimizes the Residual Sum of Squares (RSS). The Residual Sum of Squares is given by:

$$RSS = \sum_{i=1}^{n} \left(r_{i,t+1} - \beta_0 - \mathbf{x}_{i,t}^T \boldsymbol{\beta}\right)^2$$

The analytical solution for $\boldsymbol{\beta}$ is obtained by solving the first-order conditions of the optimization problem. It is given by:

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

where $\mathbf{X}$ is the design matrix of predictors, with each row representing a firm and each column representing a feature. The vector $\mathbf{y}$ represents the observed returns for all firms, and $\mathbf{X}^T\mathbf{X}$ is the Gram matrix, which captures the correlations between the predictor variables.

3.2.2 ElasticNet Regression

ElasticNet Regression addresses the limitations of Simple Linear Regression (SLR) by introducing regularization, combining the benefits of L1 (Lasso) and L2 (Ridge) penalties. The ElasticNet objective function is defined as:

$$L(\boldsymbol{\beta}) = \sum_{i=1}^{n}\left(r_{i,t+1} - \mathbf{x}_{i,t}^T\boldsymbol{\beta}\right)^2 + \alpha\left(\rho\|\boldsymbol{\beta}\|_1 + (1-\rho)\|\boldsymbol{\beta}\|_2^2\right)$$

In this model, the term $\|\boldsymbol{\beta}\|_1 = \sum_{j=1}^{p}|\beta_j|$ represents the L1 norm, which encourages sparsity in the model by driving some of the regression coefficients β_j to zero. The term $\|\boldsymbol{\beta}\|_2^2 = \sum_{j=1}^{p}\beta_j^2$ denotes the L2 norm, which shrinks the coefficients to prevent overfitting by penalizing large values. The parameter α controls the overall strength of the regularization, influencing the magnitude of both the L1 and L2 penalties. The mixing parameter ρ balances the trade-off between L1 and L2 regularization, allowing the model to adjust the relative contributions of the two penalties.

ElasticNet is particularly effective in high-dimensional settings where multicollinearity and irrelevant predictors may be present. By combining L1 and L2 regularization, it performs well even when the number of predictors exceeds the number of observations.

The ElasticNet model is typically implemented using GridSearchCV to optimize the hyperparameters α and ρ over a predefined range. Since ElasticNet is sensitive to the magnitudes of the predictor variables, it is important to scale the predictors using a method like StandardScaler before fitting the model. The best-performing model is then selected based on its validation R^2 score, which measures the proportion of variance in the response variable explained by the model.

3.3 Dimension Reduction: PCR and PLS

3.3.1 Principal Component Regression

Principal Component Regression (PCR) mitigates multicollinearity by combining Principal Component Analysis (PCA) with linear regression. PCA decomposes the feature matrix $\mathbf{X} \in \mathbb{R}^{n\times p}$ into orthogonal principal components $\mathbf{Z} \in \mathbb{R}^{n\times k}$, ranked by their explained variance. The $(k-1)$-dimensional principal components are defined as:

$$\mathbf{Z} = \mathbf{X}\mathbf{W},$$

where $\mathbf{W} \in \mathbb{R}^{p\times k}$ is the projection matrix that maximizes the variance of $\mathbf{Z}$. Components explaining at least 85% of the cumulative variance are selected to ensure dimensionality reduction while retaining predictive information. The subsequent regression model is expressed as:

$$\mathbf{y} = \mathbf{Z}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where $\boldsymbol{\beta}$ denotes the regression coefficients, and $\boldsymbol{\epsilon}$ is the residual term. By orthogonalizing features, PCR eliminates multicollinearity, thus improving model stability.

3.3.2 Partial Least Squares

Partial Least Squares Regression (PLS) extends Principal Component Regression (PCR) by incorporating target information during dimensionality reduction. PLS projects $\mathbf{X} \in \mathbb{R}^{n\times p}$ and $\mathbf{y} \in \mathbb{R}^n$ into a shared latent space, iteratively solving:

$$\max_{\mathbf{w}} \text{Cov}(\mathbf{X}\mathbf{w}, \mathbf{y})$$

where $\mathbf{w}$ represents the weight vector defining the latent variables. The latent variables $\mathbf{T}$ are constructed as:

$$\mathbf{T} = \mathbf{X}\mathbf{W}_T$$

with $\mathbf{W}_T$ maximizing the covariance between $\mathbf{X}$ and $\mathbf{y}$. Subsequent components are constrained to be orthogonal to prior ones, ensuring non-redundancy. The final regression model is expressed as:

$$\mathbf{y} = \mathbf{T}\boldsymbol{\gamma} + \boldsymbol{\epsilon}$$

where **T** are the extracted latent variables, $\boldsymbol{\gamma}$ are the regression coefficients, and $\boldsymbol{\epsilon}$ is the residual. By directly optimizing feature-target covariance, PLS enhances predictive performance, particularly when **X** and **y** exhibit strong relationships.

3.4 Generalized Linear

Linear regression models serve as a foundational tool for statistical modeling but are limited by three primary assumptions:

1. The response variable follows a normal distribution.
2. A strictly linear relationship exists between the predictor variables and the response variable.
3. The variance of the response variable is constant across all observations.

However, these assumptions rarely hold true in real-world applications. For instance, data may exhibit non-linearity, heteroscedasticity (non-constant variance), or response variables that follow distributions other than normal (e.g., binary or count data). To address these challenges, Generalized Linear Models (GLMs) extend linear regression by introducing a link function that connects the expected value of the response variable, μ_i, to a linear predictor, $x_i^\top \beta$:

$$\mu_i = x_i^\top \beta$$

$$\mathbb{E}(y_i) = f^{-1}(\mu_i)$$

Here, f is the link function, which allows GLMs to model non-linear relationships between the response and predictors while maintaining the interpretability of linear regression. Importantly, GLMs also accommodate response variables that belong to the exponential family of distributions (e.g., normal, Bernoulli, Poisson). This flexibility makes GLMs suitable for a wide range of applications, from logistic regression for binary outcomes to Poisson regression for count data.

In the theoretical framework of GLMs, linear regression and logistic regression are regarded as special cases. By bridging the gap between these cases, GLMs provide a unified approach to modeling both linear and non-linear relationships, as well as data with non-constant variance.

While GLMs extend the flexibility of linear regression, they are still limited when dealing with highly non-linear relationships. In these cases, further enhancements are necessary to capture the underlying patterns in the data. Spline regression addresses these limitations by dividing the data into multiple segments and fitting polynomial models within each segment. These segments are separated by points known as knots. For example, consider a K-term spline series expansion of the predictors:

$$g(z; \theta, \mathbf{p}(\cdot)) = \sum_{j=1}^{p} p_j(z)\theta_j$$

where $\mathbf{p}(\cdot) = (p_1(\cdot), p_2(\cdot), \ldots, p_k(\cdot))^\top$ is a vector of basis functions and θ is a $K \times N$ matrix of parameters.

To enhance the flexibility of spline regression, a series of order two can be adopted, where the basis functions include terms like:

$$\big(1, z, (z-c_1)^2, (z-c_2)^2, \ldots, (z-c_{K-2})^2\big)$$

where $c_1, c_2, \ldots, c_{K-2}$ are the knot locations. The choice of knots is critical, as they determine how the data is partitioned and how well the spline captures the underlying patterns.

One of the key strengths of spline regression lies in its ability to ensure continuity and smoothness at the knots. To achieve this, constraints are applied so that the polynomials on either side of a knot:

- Are continuous in value.
- Share consistent derivatives up to a certain order.

For instance, an m-order spline ensures that the polynomials are continuous in their value and their first $m-1$ derivatives. This smoothness is essential for maintaining the interpretability and stability of the model, especially when dealing with complex, non-linear relationships. Additionally, the smoothness property helps to avoid abrupt changes in the model's predictions, which can lead to overfitting and reduced generalization performance.

However, as spline regression increases the number of parameters to capture more nuanced patterns, the risk of overfitting grows. This necessitates the use of regularization techniques to control model complexity. To address the risk of overfitting in models with high-dimensional spline expansions, group lasso regularization is introduced. This approach penalizes the model's complexity while preserving its

ability to fit the data effectively. The group lasso penalty is defined as

$$\phi(\theta; \lambda, K) = \lambda \sum_{j=1}^{p} \left(\sum_{k=1}^{K} \theta_{j,k}^2 \right)^{1/2}$$

where λ controls the strength of the penalty, balancing model fit and complexity, and K represents the number of knots, which determines the granularity of the spline segments.

Group lasso regularization is particularly effective for spline regression because it operates on groups of coefficients associated with the spline basis functions. By shrinking some coefficients to zero, it simplifies the model without compromising its ability to capture key patterns in the data. The group lasso penalty is implemented using an accelerated gradient descent algorithm, similar to the elastic net method. This ensures computational efficiency and scalability to large datasets.

Figure 2 is a schematic of a statistical model that is commonly used to describe the structure of a Generalized Linear Model (GLM).□

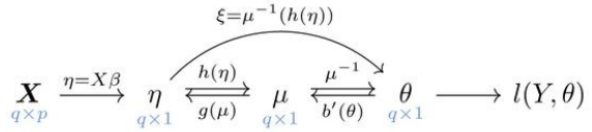


Figure 2: The Structure of GLM

Generalized linear models and spline regression represent powerful tools for statistical modeling, particularly when traditional linear regression is insufficient. By combining the flexibility of GLMs, the adaptability of spline regression, and the robustness of group lasso regularization, this framework provides a comprehensive solution for modeling complex, non-linear relationships in high-dimensional data.

3.5 Random Forests

Random Forest is an ensemble learning technique that enhances prediction accuracy by combining multiple decision trees. Each tree is trained on a randomly selected subset of the data using bootstrapping, and during each split, only a random subset of features is considered.

3.5.1 What is the CART Tree?

The Classification and Regression Tree (CART) is a decision tree algorithm used for both classification and regression tasks. The fundamental idea behind CART is to recursively partition the data into subsets, which are as homogeneous as possible with respect to the target variable. It constructs a binary tree by selecting the best split at each node based on a certain criterion. The tree is built using a recursive process that involves selecting the optimal feature and split point to minimize a particular cost function, such as Gini impurity for classification or Mean Squared Error (MSE) for regression.

1. Splitting Criterion

For classification, CART typically uses the Gini index (impurity) to evaluate the quality of a split. The Gini index for a node t is given by:

$$\text{Gini}(t) = 1 - \sum_{k=1}^{K} p_k^2$$

where p_k is the probability of class k in the node, and K is the total number of classes. The objective is to split the data such that the Gini index is minimized.

For regression, the criterion is typically the Mean Squared Error (MSE). For a node t, the MSE is calculated as:

$$\text{MSE}(t) = \frac{1}{N_t} \sum_{i \in T_t} (y_i - \bar{y}_t)^2$$

where N_t is the number of samples in the node, y_i is the actual value for observation i, and $\bar{y}_t$ is the mean of the target values in node t.

2. Tree Construction

The tree is grown by choosing the best feature and threshold at each node, such that the selected split minimizes the Gini impurity (for classification) or MSE (for regression). The process can be formalized as:

$$\text{Best Split} = \arg\min_{s} \sum_{t \in T(s)} \frac{|T_t|}{|T|} \cdot \text{Impurity}(t)$$

where $T(s)$ is the set of child nodes resulting from a split s, T_t is a subset of data points in node t, and $|T|$ and $|T_t|$ represent the number of data points in the parent and child node, respectively.

3. Stopping Criteria

The tree is grown until a stopping condition is met. Common stopping conditions include:

- The node contains fewer than a minimum number of samples.
- Further splits do not reduce the impurity by a significant amount.
- The tree reaches a specified maximum depth.

4. Prediction

For classification, once the tree is constructed, the prediction for a new sample is made by traversing the tree from the root to a leaf. The class predicted is the majority class in the leaf node.

For regression, the predicted value is the mean of the target values of the training data points in the leaf node.

3.5.2 What is the Bagging Integration Algorithm?

The Bagging (Bootstrap Aggregating) algorithm is an ensemble learning technique that improves the accuracy and robustness of machine learning models, particularly decision trees. Bagging works by combining multiple models (usually the same type) trained on different subsets of the data to reduce variance and prevent overfitting.

1.Bootstrap Sampling

A bootstrap sample D_b is generated by sampling with replacement from the original training set D. For each bootstrap sample, the size of D_b is typically the same as the original dataset D, but some samples are repeated while others are excluded.

Mathematically, the bootstrap sample D_b consists of n observations selected randomly from D with replacement:

$$D_b = \{(x_{b1}, y_{b1}), (x_{b2}, y_{b2}), \ldots, (x_{bn}, y_{bn})\}$$

where $D_b \subseteq D$ and each observation (x_{bi}, y_{bi}) is selected randomly with replacement from D.

2.Model Training

For each bootstrap sample D_b, a model $f_b(x)$ is trained. The models are trained independently using the corresponding bootstrap sample. The function $f_b(x)$ is the model trained on the bootstrap sample D_b. Thus, we have M models $f_1(x), f_2(x), \ldots, f_M(x)$, where each model $f_b(x)$ is trained on its respective bootstrap sample D_b. The models are generally of the same type (e.g., decision trees) but trained independently on different data subsets.

3.Aggregation

After training MMM models, the predictions are aggregated to make the final prediction: For classification, a majority vote is taken across all models: $\hat{y} = \text{mode}(f_1(x), f_2(x), \ldots, f_M(x))$ where mode represents the most frequent class predicted by the models. For regression, the average of the predictions is computed:

$$\hat{y} = \frac{1}{M} \sum_{b=1}^{M} f_b(x)$$

where $f_b(x)$ is the prediction of the b-th model on input x, and the final prediction is the average of all individual model predictions.

3.5.3 Constructing Random Forests

1.Bootstrap Sampling

For each tree, a random subset of the training data is selected with replacement to create a new training set, which may contain repeated instances. This is also called a bootstrap sample.

The bootstrap sample D_b for each tree is drawn from the original training set D, and is represented as:

$$D_b = \{(x_{b1}, y_{b1}), (x_{b2}, y_{b2}), \ldots, (x_{bn}, y_{bn})\}$$

where each observation (x_{bi}, y_{bi}) is selected randomly with replacement from D.

2.Feature Selection (Random Subspace Method)

Unlike in traditional decision trees, where all features are considered for splitting at each node, Random Forest introduces randomness by randomly selecting a subset of features at each node. This process is known as the random subspace method. For each node in a tree, instead of considering all p features, only a random subset of m features (where m is typically $\sqrt{p}$ or $\log_2(p)$) is evaluated for the best split. This further decorrelates the individual trees in the forest.

At each node, the model considers only mmm random features and selects the feature with the best split according to a chosen criterion (such as Gini index for classification or MSE for regression).

3.Tree Construction

Each tree in a Random Forest is built using a bootstrap sample D_b and a random subset of features. The tree is grown to its maximum depth without pruning, capturing complex patterns and leading to overfitting on the bootstrap sample. However, overfitting is controlled by averaging or voting the predictions of all trees.

The tree construction evaluates the impurity at each node and selects the best split based on the impurity measure. For node t, this is determined using p_k (class probability), N_t (samples in node), y_i (target values), and $\bar{y}_t$ (mean target value in the node).

Model testing proceeds similarly.

4.Prediction Aggregation

For classification, the final prediction is made by taking the majority vote of the individual trees:

$$\hat{y} = \text{mode}(f_1(x), f_2(x), \ldots, f_M(x))$$

For regression, the final prediction is made by taking the average of the predictions from all the trees:

$$\hat{y} = \frac{1}{M} \sum_{b=1}^{M} f_b(x)$$

where $f_b(x)$ is the prediction of the b-th tree on input x. Figure 3 illustrates the complete flowchart of the Random Forest construction process.

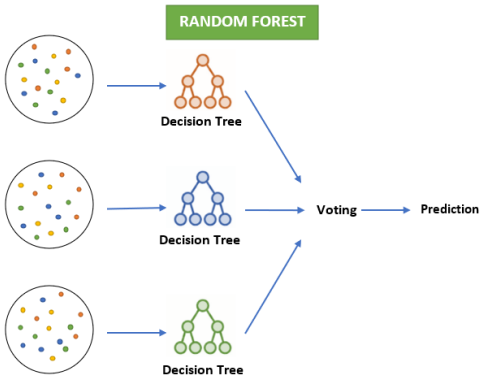


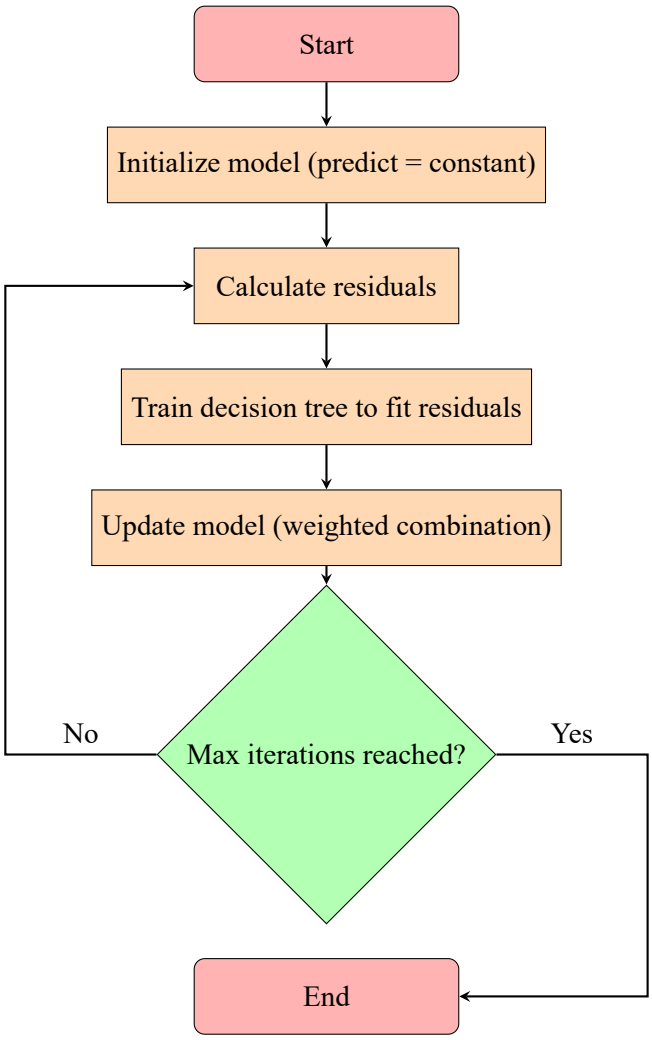
Figure 3: Random Forest Workflow: Ensemble of Decision Trees with Voting for Prediction (Luwe et al., 2023)

3.5.4 *Comparison of Random Forest with Traditional Machine Model*

As shown in Table 1. This table highlights the advantages of Random Forest compared to other machine learning algorithms. Random Forest utilizes an ensemble of decision trees, offering robustness, accuracy, and resistance to overfitting. It provides built-in variable importance for feature selection and handles missing data effectively. Additionally, it supports parallelization for faster computation, unlike many other algorithms.

3.6 Gradient Boosting Tree

Gradient Boosting Tree (GBT) is a machine learning algorithm that builds an ensemble of decision trees, each fitting the residual errors of the previous model. It updates the model by combining the new tree with the existing ensemble until a predefined number of iterations or convergence is reached.



3.6.1 *What is a Decision Tree?*

A decision tree is a supervised learning model widely used for both classification and regression tasks. Its structure mimics a hierarchical tree, where each internal node represents a decision rule based on an in-

Table 1: Comparison of Random Forest with Traditional Machine Learning Models

Feature	Random Forest	Other ML Algorithms
Ensemble Approach	Utilizes an ensemble of decision trees, combining their outputs for predictions, fostering robustness and accuracy.	Typically relies on a single model (e.g., linear regression, support vector machine) without an ensemble approach.
Overfitting Resistance	Resistant to overfitting due to the aggregation of diverse decision trees, preventing memorization of training data.	Some algorithms may be prone to overfitting, especially when dealing with complex datasets.
Variable Importance	Provides a built-in mechanism for assessing variable importance, aiding in feature selection and interpretation of influential factors.	Many algorithms may lack an explicit feature importance assessment, making it challenging to identify crucial variables.
Handling of Missing Data	Exhibits resilience in handling missing values by leveraging available features for predictions, contributing to practicality in real-world scenarios.	Other algorithms may require imputation or elimination of missing data, potentially impacting model training and performance.
Parallelization Potential	Capitalizes on parallelization, enabling the simultaneous training of decision trees, resulting in faster computation for large datasets.	Some algorithms may have limited parallelization capabilities, potentially leading to longer training times for extensive datasets.

put feature, branches correspond to the outcomes of these rules, and leaf nodes provide the final prediction, either as a discrete class label or a continuous value. The construction of decision trees involves recursively partitioning the feature space into subsets that are increasingly homogeneous with respect to the target variable. This partitioning process is guided by measures such as Gini impurity for classification tasks, defined as:

$$\text{Gini Impurity} = 1 - \sum_{i=1}^{C} p_i^2,$$

where p_i is the proportion of samples belonging to class i in a given node, and C is the total number of classes. For regression, the most commonly used criterion is the minimization of the Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y})^2,$$

where y_i is the true target value, $\hat{y}$ is the predicted value, and n is the number of observations in the node.

Although decision trees are intuitive and interpretable, they are prone to overfitting, particularly when grown to excessive depths. Overfitting occurs when the tree captures noise or minor fluctuations in the training data, thereby reducing its ability to generalize to unseen data. To mitigate this, regularization techniques such as limiting the maximum depth of the tree, setting a minimum number of samples required to split a node, or applying post-pruning strategies are often employed. Despite their limitations, decision trees form the foundation of more robust ensemble methods such as Random Forests and Gradient Boosting Trees, significantly enhancing predictive performance.

3.6.2 What is Gradient Boosting?

Gradient Boosting is a machine learning algorithm that builds predictive models by iteratively combining weak learners, typically shallow decision trees, in a stage-wise fashion. It is grounded in the principle of minimizing a predefined loss function by successively fitting new models to the negative gradient of the loss with respect to the current predictions. The

process begins by initializing the model with a constant value that minimizes the loss across the entire training set:

$$F_0(x) = \arg\min_c \sum_{i=1}^{n} L(y_i, c),$$

where $L(y, c)$ denotes the loss function, y is the target variable, and c is a constant. At each subsequent iteration m, the algorithm computes the residuals, which represent the gradient of the loss function with respect to the current model predictions:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}.$$

A new decision tree $h_m(x)$ is then trained to approximate these residuals, and the model is updated using a weighted combination of the previous model and the newly trained tree:

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x),$$

where ν is the learning rate, controlling the contribution of each tree to the overall model. The iterative process continues until a predefined number of iterations is reached or the loss function converges to a satisfactory value.

Gradient Boosting is known for its ability to achieve high predictive accuracy, making it a favored choice in many applications, including financial modeling, healthcare analytics, and natural language processing. However, its computational cost can be significant, particularly for large datasets. Modern implementations, such as XGBoost, LightGBM, and CatBoost, address these challenges by introducing innovations such as histogram-based splitting, parallelization, and improved handling of categorical variables.

3.6.3 What is Boosting Method?

Boosting is a powerful ensemble learning technique designed to convert a set of weak learners into a single strong learner. Unlike bagging methods, where weak learners are trained independently, boosting trains them sequentially, with each learner correcting the errors made by its predecessors. This approach relies on a combination of additive modeling and weighted loss minimization. Formally, the boosted model can be expressed as:

$$F(x) = \sum_{m=1}^{M} \alpha_m h_m(x),$$

where $h_m(x)$ represents the m-th weak learner, α_m is its weight, and M is the total number of learners. Each learner focuses on examples that were misclassified or poorly predicted by the previous model. As shown in Figure 4, Boosting works by sequentially adjusting residuals and training weak learners to reduce errors in successive iterations. For instance, in AdaBoost (Adaptive Boosting), the weights of training samples are updated as follows:

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp\left(-\alpha_m \cdot y_i \cdot h_m(x_i)\right),$$

where $w_i^{(m)}$ is the weight of sample i at iteration m, y_i is the true label, and α_m is the weight of the m-th learner determined by its accuracy. Samples that are misclassified receive higher weights, forcing the next learner to focus more on these difficult examples.

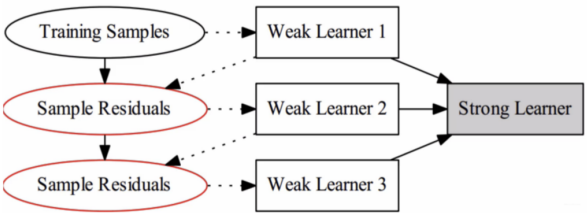


Figure 4: The Boosting process: Sequentially training weak learners to form a strong learner (Wang, 2020)

Gradient Boosting, a specific instance of boosting, extends this framework by directly optimizing a differentiable loss function using gradient descent. Gradient Boosting builds upon the Boosting framework shown in Figure 4, replacing the weight adjustment with gradient-based optimization of the loss function. Boosting methods, while capable of achieving remarkable accuracy, are sensitive to hyperparameters such as the learning rate and number of iterations, and they may suffer from overfitting if the learners are excessively complex or the algorithm is run for too many iterations. Nonetheless, boosting remains one of the most successful techniques in supervised learning, underpinning many state-of-the-art algorithms.

3.7 LightGBM

LightGBM (Light Gradient Boosting Machine) was developed to address the inefficiencies encountered by traditional GBDT (Gradient Boosting Decision Tree) algorithms when handling massive datasets. Traditional GBDT models face significant challenges in both memory usage and computational speed, especially when applied to industrial-scale problems. LightGBM introduces innovative techniques that allow it to process large datasets efficiently, making it a practical choice for real-world machine learning tasks.

Table 2 below is a comparison of LightGBM and Other GBDT Models.

This part focuses on the key algorithms and features that make LightGBM an upgrade over other implementations of GBDT, such as XGBoost. These innovations include the Histogram Algorithm, Leaf-wise Growth Strategies, One-sided Gradient Sampling, and Exclusive Feature Bundling, among others.

3.7.1 Histogram Algorithm

The Histogram Algorithm is a core component of LightGBM that optimizes the process of calculating split points in decision trees. By discretizing continuous floating-point features into bins, the algorithm reduces the memory and computational costs associated with pre-sorted algorithms.

The key steps include:

- Discretizing feature values into integer bins of fixed width.
- Building histograms for leaf nodes by accumulating data into bins.
- Using histogram subtraction to quickly compute the histogram for sibling nodes, effectively halving the computation time.

This approach not only accelerates the training process but also minimizes memory usage by representing feature values as discrete bins instead of continuous values.

3.7.2 Leaf-Wise Growth Strategies with Depth Constraints

Traditional tree growth algorithms, such as those in XGBoost, adopt a level-wise strategy, which grows all nodes at the same depth before moving deeper into the tree. While this reduces overfitting, it increases computation and redundancy in the search for optimal splits. LightGBM uses a leaf-wise strategy, which prioritizes the leaf with the highest potential gain for splitting. To avoid overfitting, a maximum depth constraint is applied. This strategy achieves better accuracy while maintaining efficiency and reducing unnecessary splits.

The following illustrates two key innovations in LightGBM: the Histogram Algorithm and the Leaf-wise Tree Growth strategy, as shown in Figure 5. These two methods significantly improve the efficiency and accuracy of LightGBM, making it a powerful tool for handling large-scale datasets.

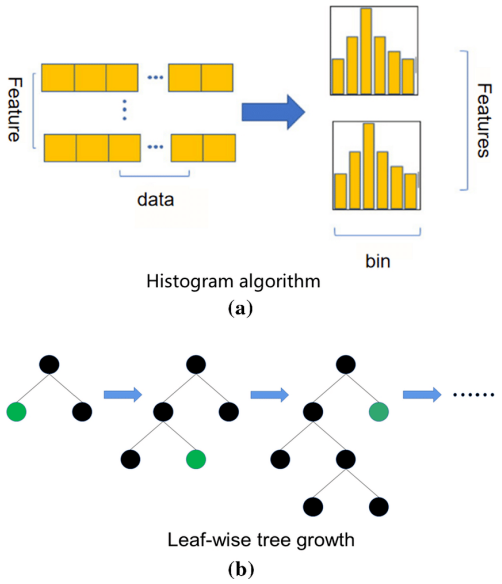


Figure 5: Two key innovations in LightGBM (Yan, Li, Cheng, Chen, & Wu, 2022)

3.7.3 One-Sided Gradient Sampling Algorithm

LightGBM introduces the Gradient-based One-Side Sampling (GOSS) algorithm to optimize gradient-based training. GOSS improves computational efficiency by:

1. Retaining instances with large gradient values (important for learning) while randomly sampling from instances with small gradient values.
2. Adjusting the distribution of the retained gradients to maintain their statistical properties.

This method strikes a balance between training accuracy and computational efficiency, making LightGBM faster without sacrificing performance. The pseudo-code for Histogram-based Algorithm and Gradient-based One-Side Sampling is as follows:

Table 2: Comparison of LightGBM and Other GBDT Models

Feature	LightGBM	Other GBDT Models (e.g., XGBoost)
Tree Growth Algorithm	Leaf-wise growth strategy with depth constraints	Level-wise growth strategy
Split Point Calculation	Histogram Algorithm	Pre-sorted algorithm
Gradient Sampling	GOSS	Uniform gradient sampling
High-dimensional Data Handling	Exclusive Feature Bundling (EFB)	No explicit feature bundling
Memory Usage	Lower memory usage	Higher memory usage
Training Speed	Faster	Slower
Accuracy	Higher accuracy with leaf-wise splits and GOSS	Can lead to overfitting with level-wise growth

Algorithm 1 Histogram-based Algorithm

Input:

I: Training data
d: Maximum tree depth
m: Number of features

Procedure:

Initialize:
  $nodeSet \leftarrow \{0\}$
  $rowSet \leftarrow \{\{0, 1, 2, \dots\}\}$
for each depth level $i = 1$ to d **do**
  for each $node$ in $nodeSet$ **do**
    $usedRows \leftarrow rowSet[node]$
    for each feature $k = 1$ to m **do**
      $H \leftarrow$ new Histogram()
      for each data point j in $usedRows$ **do**
        $bin \leftarrow I.f[k][j].bin$
        $H[bin].y \leftarrow H[bin].y + I.y[j]$
        $H[bin].n \leftarrow H[bin].n + 1$
      end for
      Determine the best split using histogram H
    end for
  end for
  Update $rowSet$ and $nodeSet$ based on the best split points
end for

Algorithm 2 Gradient-based One-Side Sampling (GOSS)

Input:

I: Training data
d: Number of boosting iterations
a: Sampling ratio for large gradient instances
b: Sampling ratio for small gradient instances
$loss$: Loss function
L: Weak learner

Procedure:

Initialize:
  $models \leftarrow \{\}$
  $fact \leftarrow \frac{1-a}{b}$
  $topN \leftarrow a \times \text{len}(I)$
  $randN \leftarrow b \times \text{len}(I)$
for each iteration $i = 1$ to d **do**
  $preds \leftarrow models.predict(I)$
  $g \leftarrow loss(I, preds)$
  $w \leftarrow \{1, 1, \dots\}$
  $sorted \leftarrow$ GetSortedIndices(abs(g))
  $topSet \leftarrow$ sorted[1 : topN]
  $randSet \leftarrow$ RandomPick(sorted[topN : len(I)], randN)
  $usedSet \leftarrow topSet + randSet$
  w[randSet]$\times =$ fact
  $newModel \leftarrow L(I[\text{usedSet}], -g[\text{usedSet}], w[\text{usedSet}])$
  $models.append(newModel)$
end for

3.7.4 The Greedy Bunding and Merge Exclusive Features Algorithms

LightGBM also incorporates the Greedy Bundling and Merge Exclusive Features algorithms within the framework of Exclusive Feature Bundling (EFB) to enhance its ability to handle high-dimensional datasets more efficiently. These two algorithms work in tandem to address the challenges posed by the large number of features in high-dimensional data, particularly when many of these features are sparse and mutually exclusive.

The Greedy Bundling Algorithm identifies features that rarely take non-zero values simultaneously and groups them into bundles while minimizing conflicts. This is achieved through the construction of a conflict graph $G = (V, E)$, where V represents the set of features and E represents edges weighted by the conflict ratio:

$$w_{ij} = \frac{\text{Conflicts}(f_i, f_j)}{\min(\text{Usage}(f_i), \text{Usage}(f_j))},$$

where $\text{Conflicts}(f_i, f_j)$ is the number of rows where f_i and f_j are non-zero simultaneously, and $\text{Usage}(f_i)$ is the number of rows where f_i is non-zero. The goal is to minimize the total conflict cost:

$$\text{Cost} = \sum_{B_k} \sum_{(f_i, f_j) \in B_k} w_{ij},$$

where B_k represents each bundle.

The Merge Exclusive Features Algorithm takes these bundles and efficiently merges the features within them. For each bundle B_k, the total bin range is computed as:

$$\text{BinRange}_k = \sum_{f_i \in B_k} \text{NumBins}(f_i),$$

where $\text{NumBins}(f_i)$ is the number of bins for feature f_i. Each feature's bins are mapped into a unified bin space using:

$$\text{NewBin}[i] = \text{BinOffset}(i) + \text{BinValue}(f_i),$$

where $\text{BinOffset}(i)$ is the cumulative bin offset of the feature within the bundle, and $\text{BinValue}(f_i)$ is the original bin value.

The specific implementations of these algorithms are demonstrated in the following pseudo-code.(Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., & Liu, T. Y. (2017))

Algorithm 3 Greedy Bundling

Input:
F: Features
K: Max conflict count

Procedure:

Construct graph G
$searchOrder \leftarrow G.sortByDegree()$
$bundles \leftarrow \{\}$, $bundlesConflict \leftarrow \{\}$
for i **in** $searchOrder$ **do**
  $needNew \leftarrow$ True
  for $j = 1$ **to** len$(bundles)$ **do**
    $cnt \leftarrow$ ConflictCnt$(bundles[j], F[i])$
    if $cnt +$ bundlesConflict$[i] \leq K$ **then**
      $bundles[j].add(F[i])$
      $needNew \leftarrow$ False
      break
    end if
  end for
  if $needNew$ **then**
    Add $F[i]$ as a new bundle to $bundles$
  end if
end forOutput: $bundles$

Algorithm 4 Merge Exclusive Features

Input:
$numData$: Number of data
F: One bundle of exclusive features

Procedure:

$binRanges \leftarrow \{\}$, $totalBin \leftarrow 0$
for f **in** F **do**
  $totalBin \leftarrow totalBin + f.numBin$
  $binRanges.append(totalBin)$
end for
$newBin \leftarrow$ new Bin$(numData)$
for $i = 1$ **to** $numData$ **do**
  $newBin[i] \leftarrow 0$
  for $j = 1$ **to** len(F) **do**
    if $F[j].bin[i] \neq 0$ **then**
      $newBin[i] \leftarrow F[j].bin[i] + binRanges[j]$
    end if
  end for
end forOutput: $newBin, binRanges$

3.8 Neural Networks

Finally, we will introduce Neutral Networks with nonlinear method. Neural network (NN) is a complex network system formed by a large number of simple processing units (called neurons) widely connected. The main advantage of neural networks is that they can flexibly process various types of data such as images, texts, time series, and can express various complex nonlinear relationships. But at the same time, the complexity of the network structure also makes the neural network one of the most opaque, unexplainable and parameterized machine learning tools.

Generally, neural network models can always be divided into two categories according to whether the input data information will be fed back: feedforward neural network and feeedback neural network. Feedforward neural network feedforward means that The information flow direction of the feedforward neural network is from input to output and it flows in one direction, so the amount of calculation is small. In addition, there is no feedback in the whole network. Feedback neural network is exactly the opposite. Information transmission in the network can be multi-directional, and neurons at all levels can receive signals from other neurons as well as their own feedback signals. This means that neural networks are very similar to neurons in organisms, and information is transmitted between different neurons through the "signal" of the previous layer to the next layer. To better describe a typical feedforward neural network, we take the following figure as an example, which has only one input layer, one hidden layer, and an output layer style of the predicted result:

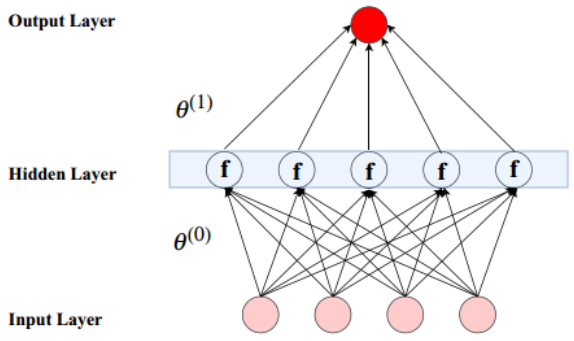


Figure 6: Neural Network Structure Diagram

In Figure 6, the input layer flexibly outputs the value of the prediction layer through the nonlinear activation function in the hidden layer and the weights between each neuron. For example, the output of the second neuron in the hidden layer can be expressed as:

$$x_2^{(1)} = f\left(\theta_{2,0}^{(0)} + \sum_{j=1}^{4} z_j \theta_{2,j}^{(0)}\right)$$

Finally, we can get the prediction result of the linear combination of each neuron in the hidden layer, which is the result of the output layer:

$$g(z;\theta) = \theta_1^{(0)} + \sum_{j=1}^{5} x_j^{(1)} \theta_j^{(1)}$$

There are many options for building neural networks, including the number of hidden layers, the number of neurons in each layer, and different weights. But it is clear from the above images that the focus is still on the hidden layers if you want to obtain flexible nonlinear predictor functions. However, as a processing layer that can integrate multiple layers of nonlinear factors, the hidden layer must consider maintaining stable prediction results under such complex interactions. If it is not possible to ensure a stable recursive calculation as the number of layers and parameters increases, then the neural network is likely to experience a "black box effect" or gradient explosion.

Therefore, we need to make reasonable considerations for the hidden layers and neurons of the neural network, which means that it must not only have a good predictive performance, but also cannot have an extremely complex model. So this requires us to use the cross-validation method to select a qualified network architecture. However, since it is not realistic and unnecessary to find the best network through countless architectures, we decided to build the network architecture first and then evaluate it to improve the predictive performance of our neural network.

Let's consider an architecture with up to 5 hidden layers. According to the geometric pyramid rule (see Masters 1993), we choose the number of neurons corresponding to these 5 hidden layers: NN1 (only one hidden layer) has 32 neurons, NN2 (two hidden layers) has 32, 16 neurons, NN3 (three hidden layers) has 32, 16, 8 neurons, NN4 (four hidden layers) has 32, 16, 8, 4 neurons, and NN5 (five hidden layers) has 32, 16, 8, 4, 2 neurons. After finding a suitable

activation function, we can test the predictive performance of neural networks with different hidden layers.

After comprehensive consideration, the activation function we chose is the linear unit (ReLU):

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

Compared with other activation functions such as sigmoid, hyperbolic tangent, and softmax, ReLU has a faster convergence speed and less computational complexity, making it more suitable for large-scale neural networks. It can also effectively alleviate the gradient vanishing problem and accelerate the training process.

So we finally have the following general formula for neural networks:

$$x_k^{(l)} = \text{ReLU}\left(x^{(l-1)'}\theta_k^{(l-1)}\right)$$

where $K^{(l)}$ represents the number of neurons in each layer $l = 1, 2, \ldots, L$, and $x_k^{(l)}$ represents the output of the k-th neuron in the l-th layer. Thus, the total output vector of the l-th layer is

$$x^{(l)} = \left(1, x_1^{(l)}, \ldots, x_k^{(l)}\right)$$

where the constant term 1 is the bias term. The input layer vector can also be represented as

$$x^{(l)} = \left(1, z_1^{(l)}, \ldots, z_k^{(l)}\right)$$

From this, we obtain the final output result:

$$g(z;\theta) = x^{(L-1)'}\theta^{(L-1)}$$

Here, each hidden layer contains $K^{(l)}$ weight parameters, and there are $1 + K^{(L-1)}$ weight parameters in total.

In the model, we also add penalty terms through L1/L2 regularization to limit the weight size and prevent the model from overfitting. However, as we mentioned earlier, the neural network model is highly nonlinear and non-convex, so we also need to introduce another more stable optimization method, namely stochastic gradient descent (SGD), to train our model. The core idea of SGD is to iteratively update the model parameters by calculating the gradient of a random subset of data in the objective function, so that the loss function is gradually reduced, thereby optimizing the performance of the model. Due to its randomness, SGD can jump out of local optima and is more likely to find the global optimal solution. However, to make up for its shortcomings, such as excessive and cumbersome calculations, we can also consider adding the Adam training method to strike a balance between efficiency and accuracy.

4 Empirical Study

4.1 Dataset

The study utilizes a set of firm characteristics that are commonly identified as potential predictors of stock returns. These include valuation and growth metrics, such as the book-to-market ratio (BM), asset growth (AGR), and capital expenditures (CAPEX), which provide insights into a firm's financial health and growth potential. Profitability indicators, including return on equity (ROEQ), operating profitability (OPERPROF), and earnings-to-price ratio (EP), reflect a firm's financial performance. Market dynamics are captured through momentum measures (e.g. MOM1M, MOM12M) and volatility indicators (e.g., IDIOVOL, RETVOL), which reflect trends and risk exposure. Liquidity and trading activity are measured using variables such as dollar volume (DOLVOL), turnover (TURN), and bid-ask spread (BASpread), which indicate the ease of trading and market depth. These features were selected based on their empirical relevance in predicting stock returns, as established in prior literature.

4.1.1 Data Collection

Data for this study is sourced from CRSP, Compustat, and I/B/E/S, covering U.S. firms from January 1980 to December 2014. CRSP provides stock returns, Compustat offers financials, and I/B/E/S supplies analyst estimates. We selected 94 firm characteristics based on prior literature. Missing values were imputed using the monthly mean for each stock, following Dacheng et al. (2020). Microcap stocks, defined as those below the 20th percentile of NYSE market capitalization, were excluded to avoid bias. The final dataset contains 1,933,898 firm-month observations from firms listed on the NYSE, AMEX, and NASDAQ.

4.1.2 Data Processing

In this study, we employed a range of data preprocessing methods to ensure the integrity of the dataset and the robustness of the empirical analysis. For handling missing values, we followed the approach outlined by Dacheng et al. (2020), which suggests, "Another issue is missing characteristics, which we replace with the cross-sectional median at each month for each stock." Based on this approach, we applied the following strategies:

For numerical variables such as betasq, chmom, dolvol, idiovol, mve1, bm, and RET, missing values were imputed using the group mean for each month. If an entire group was missing, the missing values were set to zero. For skewed variables such as age, we used median imputation to mitigate the impact of extreme values. Categorical variables like sic2 and datadate had missing values filled with the mode to maintain consistency across the dataset. For time-series variables like fiscal year (fyear), missing values were forward-filled to preserve the temporal structure. Momentum-related variables such as mom1m, mom6m, mom12m, and mom36m were filled with the mean of valid observations within the same month to ensure temporal consistency. Rows with missing gvkey values, which are essential for uniquely identifying firms, were excluded from the dataset. For data normalization, we applied the z-score standardization formula:

$$z = \frac{x - \mu}{\sigma}$$

where x is the raw data, μ is the mean, and σ is the standard deviation. Additionally, for variables with high volatility, such as std_dolvol and std_turn, standardization was performed to retain their variability and ensure consistency in the data.

These preprocessing techniques ensured the completeness of the dataset while minimizing potential biases introduced by missing data. All imputation procedures were consistent with commonly used data treatment practices in finance, providing a solid foundation for subsequent regression analysis and machine learning modeling.

4.1.3 Data Description

The dataset spans January 1980 to December 2014, containing stock returns from CRSP, financial data from Compustat, and analyst forecasts from I/B/E/S. It includes 94 firm characteristics related to profitability, growth, valuation, and market dynamics. Missing data were imputed using the monthly mean, and microcap stocks were excluded. After cleaning, the dataset consists of 1,933,898 firm-month observations from NYSE, AMEX, and NASDAQ-listed firms.

4.2 Expected Outcomes

This paper conducts empirical research on the U.S. stock market through a series of models, hoping to achieve the following research objectives: Identify key company characteristics that are most predictive of stock returns.

Compare the prediction results of each model and find the model with the highest stock return prediction accuracy.

Regression-based Models: We expect that the regression-based models can provide us with a baseline performance for our study by establishing a simple linear relationship, while also being able to clearly explain the importance of each factor and the predicted results.

Tree-based Models: We expect that the tree-based model can capture nonlinear relationships based on the regression model, be robust to noise and outliers in the data of this study, and improve the generalization ability of the model.

Neural Network Models: In this stock prediction model, the expectations of the neural network and the tree model are similar. It is hoped that the neural network model can simulate highly complex nonlinear relationships and improve its adaptability to large-scale data in this study through the characteristics of automatic learning.

4.3 Model Performance

4.3.1 Regression-Based Models

In this study, we first employed the Ordinary Least Squares (OLS) regression model as a baseline to assess its effectiveness in predicting asset returns. The R^2 values for the OLS model were 0.0463 for the training set and 0.0451 for the test set, indicating that the model's predictive capability is relatively weak, explaining only approximately 4.6% of the variance in asset returns. This result suggests that the OLS model failed to capture the complex relationships within the asset return data, limiting its applicability in this dataset.

To improve generalization, we applied the Elastic-

Net regression model, combining L1 and L2 regularization to reduce overfitting. However, its R^2 value was only 0.0255, worse than the OLS model on both the training and test sets. This suggests that excessive regularization may shrink the coefficients too much, preventing the model from capturing key features and causing underfitting.

Next, we tested the Generalized Linear Model (GLM) with spline transformations and Group Lasso regularization to model nonlinear relationships. The GLM achieved an R^2 of 0.0512 on the training set and 0.0497 on the test set. While it slightly improved over OLS and ElasticNet, its performance remained limited, and the use of nonlinear transformations did not significantly boost predictive accuracy.

Table 3: Comparative Model Performance

Models	**OLS**	**ENet**	**GLM**
Train R^2	0.0463	0.0255	0.0512
Test R^2	0.0451	0.0255	0.0497

In addition, we applied two dimensionality reduction techniques—Principal Component Regression (PCR) and Partial Least Squares (PLS)—to improve predictive performance. In the PCR model, although we selected principal components with a cumulative variance contribution greater than 85%. As shown in Figure 7, the results showed that using as many as 65 principal components did not effectively reduce model complexity, and no significant improvement in prediction was observed. Therefore, the PCR model did not provide the expected enhancement.

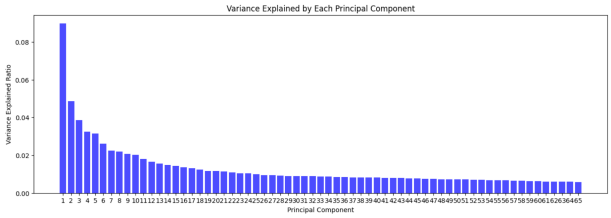


Figure 7: Variance Explained by Each Principal Component

For the PLS model, we selected 3 to 6 principal components for modeling, and the R² values for both the training and validation sets remained at 0.023, showing consistent performance. The results are summarized in Table 4. While PLS may help to capture underlying structures in the data in some cases, its overall predictive accuracy remained low, and the model did not show significant improvement.

Table 4: PLS Model Performance for Selected Principal Components

Metrics	**PLS_3PC**	**PLS_4PC**	**PLS_5PC**	**PLS_6PC**
Train R^2	0.018605	0.021122	0.022965	0.022965
Test R^2	0.018532	0.020235	0.021544	0.022432

Here comes the results analysis:

The performance of all regression models (OLS, ElasticNet, GLM) and dimensionality reduction methods (PCR and PLS) fell short of expectations. While the GLM made some progress by capturing more complex relationships using nonlinear transformations and Group Lasso regularization, its overall explanatory power remained weak. The R² values were consistently low, indicating that even with advanced techniques, the models struggled to accurately predict the asset returns. The results suggest that these models were unable to effectively capture the underlying patterns in the data.

Despite the relatively stable R² values between the training and test sets, which indicate that overfitting was not an issue, the models' performance highlights their significant limitations. The inability to achieve higher R² values in both sets points to a fundamental challenge in modeling asset returns using the chosen methods. This suggests that additional features, more advanced techniques, or alternative modeling approaches may be necessary to better capture the dynamics of asset returns and improve predictive accuracy.

4.3.2 Tree-Based Models

At random forest, as shown in Table 5, in our tests, as the number of trees increased from 100 to 400, 350 trees resulted in the smallest accuracy gap between the training and test sets, with less overfitting. `random_state = 42` ensures experiment reproducibility, keeping data splits and tree construction consistent. `min_samples_split = 20` requires at least 20 samples to split a node, controlling tree growth and reducing overfitting. `min_samples_leaf = 30` ensures each leaf has at least 30 samples, preventing small leaves that overfit noise. We tested various `max_features` settings, including `log2` and `n_features`. While `max_features = 'log2'` reduced computation, it limited tree learning, increasing bias.

`max_features = n_features` increased computation and overfitting. We chose `max_features = 'sqrt'` as it balanced efficiency and model generalization, reducing overfitting while working well with large datasets.

Table 5: Parameters of RF Model

Parameter	Value
Number of Trees	350
Max Features	Sqrt
Random State	42
min_samples_split	20
min_samples_leaf	30

The depth of the tree directly affects the model's complexity. A depth that is too large can lead to overfitting, while a depth that is too shallow may fail to capture the patterns in the data. We chose a maximum depth of 10, and after multiple experiments, as shown in Table 6, we found that it effectively captures the features in the data at a lower complexity while avoiding overfitting.

Table 6: Performance Comparison of Random Forest Models with Different Max Depth

max_depth (MD)	MD=6	MD=7	MD=8	MD=9	MD=10
R^2 on train	0.2104	0.2505	0.2978	0.3153	0.3317
R^2 on test	0.2676	0.2680	0.2720	0.2710	0.2726

The performance of the Gradient Boosting Decision Tree (GBDT) model was evaluated using several metrics, including the R^2 values for both the training and test sets, Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE). After hyperparameter tuning, the best parameter combination was obtained: $learning\_rate = 0.2$, $max\_depth = 5$, and $n\_estimators = 200$.

As shown in Table 7, the R^2 value for the training set was 0.7778, indicating that the model was able to capture a significant amount of variance in the training data, suggesting a good fit. However, the R^2 value for the test set was only 0.4618, which indicates some overfitting and reveals a weaker generalization ability of the model on unseen data. This suggests that while the model performs well on the training data, its performance deteriorates when applied to new data.

In terms of error metrics, the MAE, MSE, and RMSE for the training set were 0.0309, 0.0017, and 0.0411, respectively, reflecting high prediction accuracy on the training data. However, for the test set, the MAE, MSE, and RMSE values increased to 0.0412, 0.0039, and 0.0624, respectively, further confirming the model's reduced generalization ability on unseen data.

Table 7: Model performance metrics for GBDT

Metric	Training Set	Test Set
R^2	0.7778	0.4618
MAE	0.0309	0.0412
MSE	0.0017	0.0039
RMSE	0.0411	0.0624

Figure 8 and Figure 9 illustrate the comparison between the true values and the predicted values for the training and testing datasets, respectively. The green points represent the true values, while the red points represent the predicted values. In the training set (Figure 8), most of the predicted values closely match the true values, indicating that the model performs well in capturing the patterns of the training data. However, in the testing set (Figure 9), while the majority of predictions still align with the true values, there are noticeable deviations, particularly in regions with outliers or extreme values. These differences suggest that the model exhibits signs of overfitting, as its generalization performance on unseen data is weaker. These visualizations provide clear evidence of the model's strengths and weaknesses in handling both the training and testing datasets.

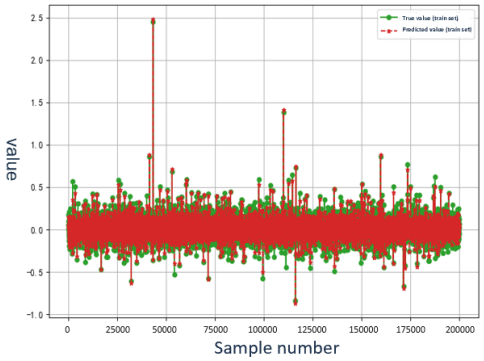


Figure 8: Comparison of Training Set True Values and Predicted Values

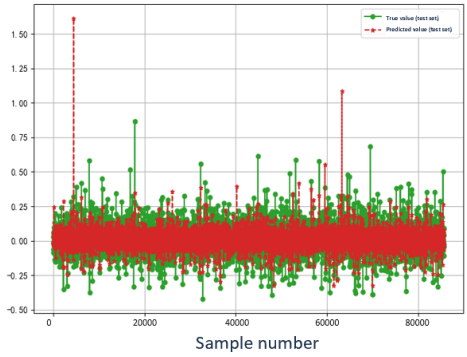


Figure 9: Comparison of Testing Set True Values and Predicted Values

At LightGBM, we made several adjustments to the hyperparameters using the validation set we divided. Finally, we found that when `learning_rate` is set to 0.015, `n_boost_round` is set to 500, and `subsample` is kept at default, the LGB model corresponding to different `max_depth` values achieves stable out-of-sample R^2 values. Among these, the LGB model with `max_depth` = 8 achieves the best out-of-sample R^2 of 0.852616 (Table 8). The results achieved by LightGBM far exceeded our expectations.

This shows that increasing the `max_depth` parameter improves the model's ability to fit the training data while maintaining generalization on the test set. During the training of the model, thanks to the engineering optimizations of the LightGBM framework, we also observed high computational efficiency. The model was trained significantly faster compared to traditional boosting models, with a speed improvement of approximately 30-40 times.

Table 8: Performance Comparison of LightGBM Models with Different Max Depth

	MD=5	**MD=6**	**MD=7**	**MD=8**
Train R^2	0.674218	0.770025	0.813287	0.853618
Test R^2	0.668965	0.767222	0.811184	0.852616

Here comes the results analysis:

The LightGBM model performed the best, achieving the highest R^2 value (0.8526) on the test set and significantly improving training speed, making it more efficient than both Random Forest and GBDT. While Random Forest performed well in avoiding overfitting and balanced the performance between the training and test sets with appropriate configurations, its computational efficiency was not as high as LightGBM. Although GBDT performed well on the training set, it showed clear overfitting on the test set and had weaker generalization ability. Overall, LightGBM outperformed the other two models in both accuracy and efficiency.

4.3.3 Neural Network Models

From the previous regression models, we can observe that, except for Enet, which does not perform as well, the other regression models show good prediction performance. We can also explore the possibility of using neural network models for stock return prediction. The R^2 values for the five neural network models mentioned above are shown in Table 9:

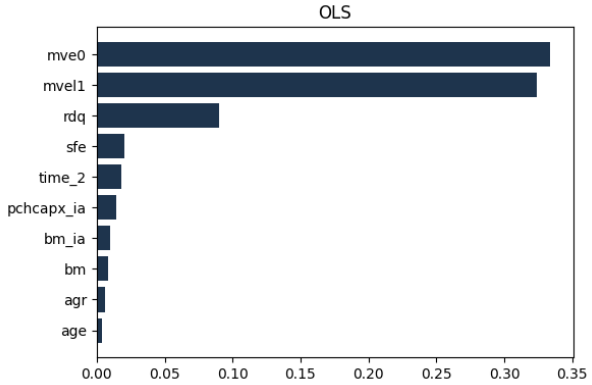
Table 9: Model Performance for Different Neural Network Models

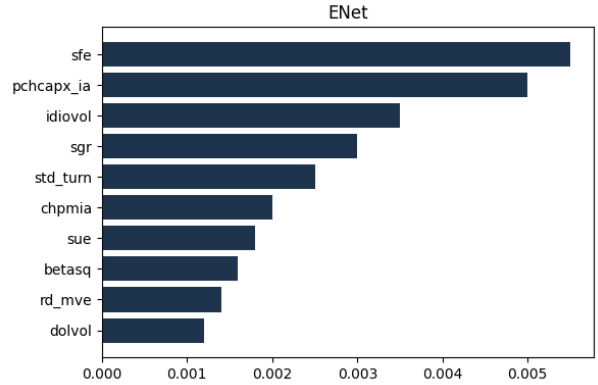
Model	R^2
NN1	0.0341
NN2	0.0433
NN3	0.0412
NN4	0.0430
NN5	0.0313

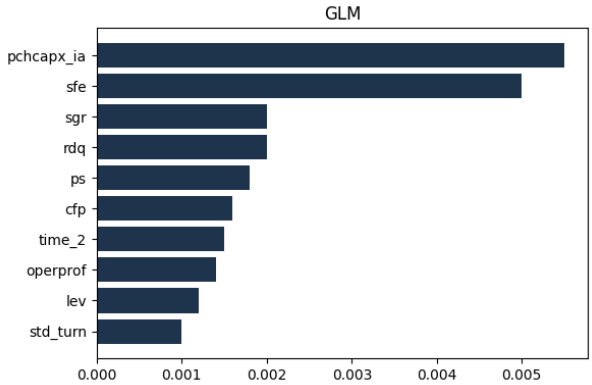
From this table, we can observe that the performance of the neural network model exhibits some volatility. Under the SGD method, the NN2 model performs best with an R^2 value of 0.043. In contrast, the R^2 values for the NN1 and NN5 models are relatively lower, at 0.034 and 0.031, respectively. However, as the number of hidden layers in the model increases, the fitting performance does not show an upward trend as expected. Therefore, we can conclude that, under the stochastic gradient method, more layers are not always better. It is necessary to find the most appropriate parameters and number of hidden layers to achieve the desired results. Additionally, we can also try using the ADAM optimizer or other methods, as they may yield different results.

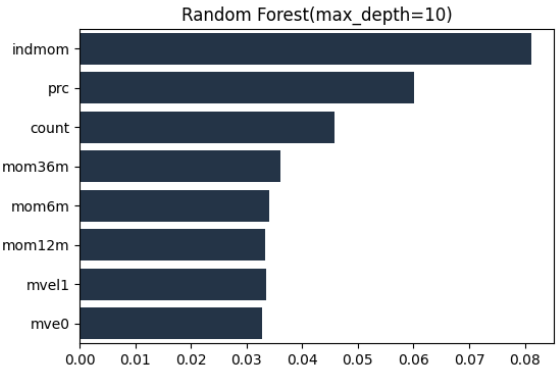
4.4 Feature Importance

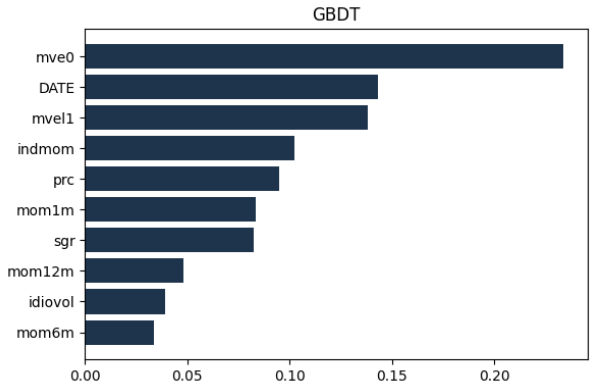
In this section, we evaluate the feature importance across different models. The results, highlighting the top 10 most influential features for each model, are presented following:

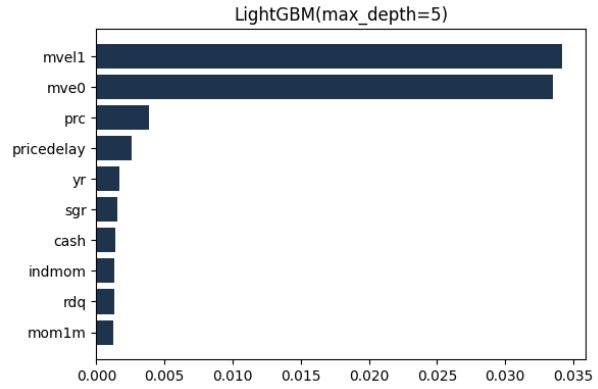

OLS
mve0
mvel1
rdq
sfe
time_2
pchcapx_ia
bm_ia
bm
agr
age
0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35

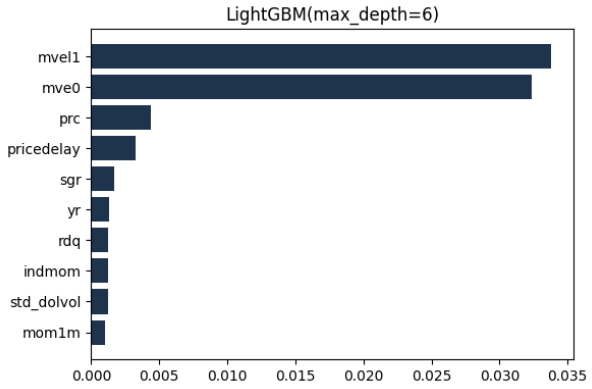

ENet
sfe
pchcapx_ia
idiovol
sgr
std_turn
chpmia
sue
betasq
rd_mve
dolvol
0.000 0.001 0.002 0.003 0.004 0.005

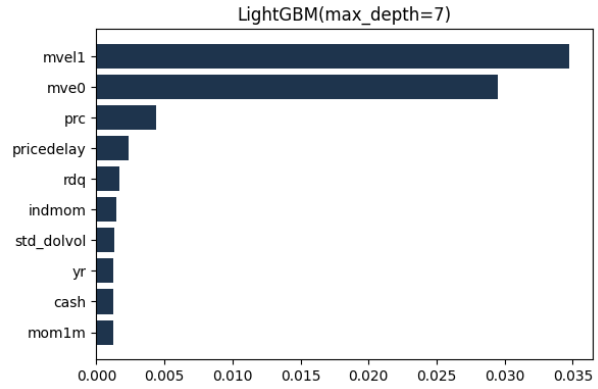

GLM
pchcapx_ia
sfe
sgr
rdq
ps
cfp
time_2
operprof
lev
std_turn
0.000 0.001 0.002 0.003 0.004 0.005

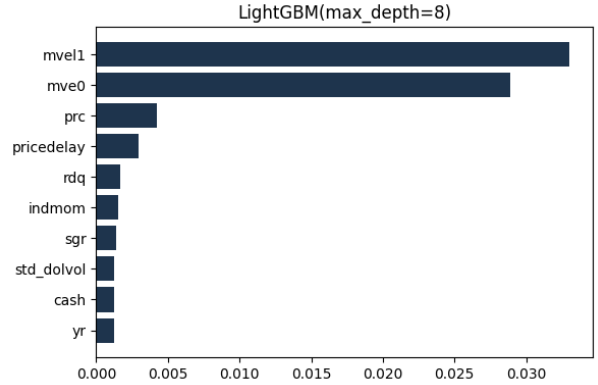

Random Forest(max_depth=10)
indmom
prc
count
mom36m
mom6m
mom12m
mvel1
mve0
0.00 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08

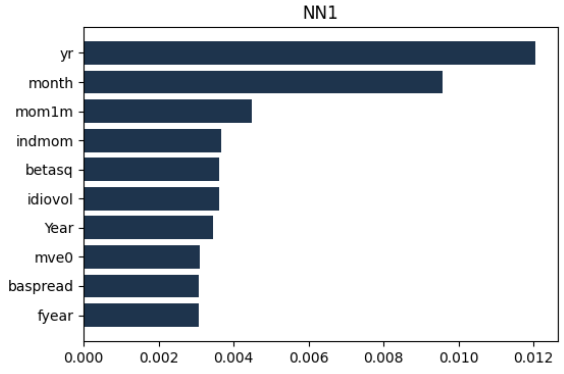

GBDT
mve0
DATE
mvel1
indmom
prc
mom1m
sgr
mom12m
idiovol
mom6m
0.00 0.05 0.10 0.15 0.20

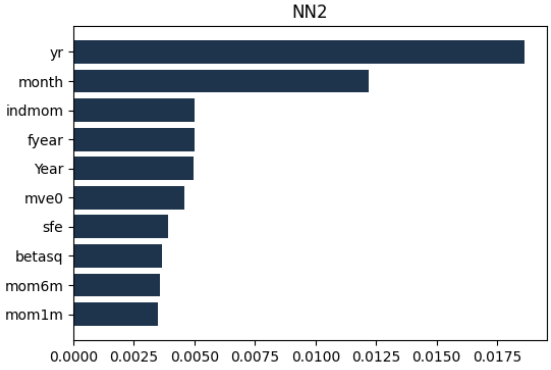

LightGBM(max_depth=5)
mvel1
mve0
prc
pricedelay
yr
sgr
cash
indmom
rdq
mom1m
0.000 0.005 0.010 0.015 0.020 0.025 0.030 0.035

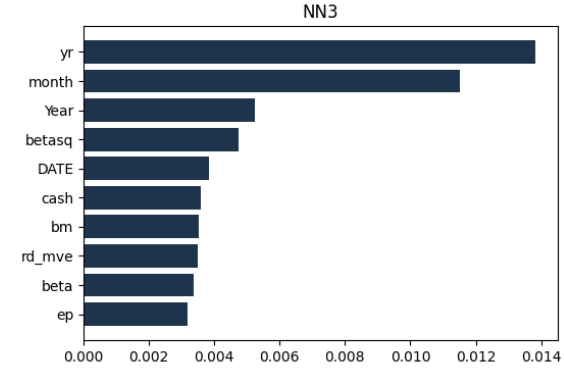

LightGBM(max_depth=6)
mvel1
mve0
prc
pricedelay
sgr
yr
rdq
indmom
std_dolvol
mom1m
0.000 0.005 0.010 0.015 0.020 0.025 0.030 0.035

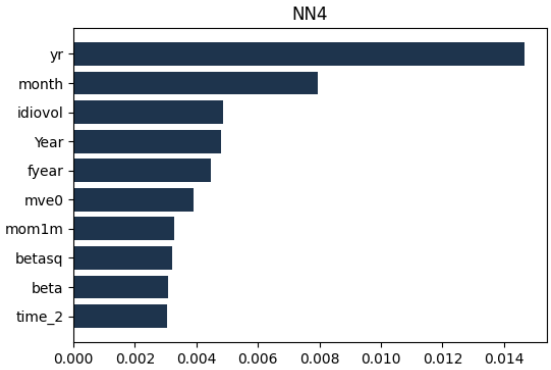

LightGBM(max_depth=7)
mvel1
mve0
prc
pricedelay
rdq
indmom
std_dolvol
yr
cash
mom1m
0.000 0.005 0.010 0.015 0.020 0.025 0.030 0.035

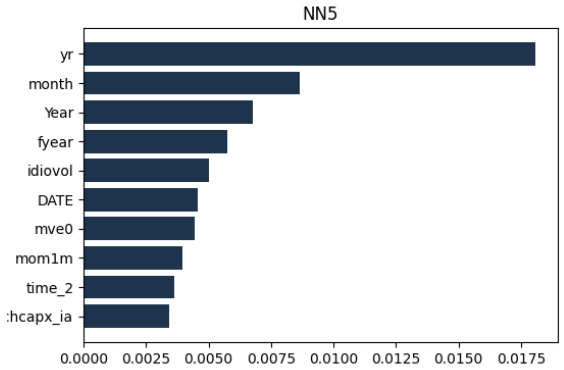

LightGBM(max_depth=8)
mvel1
mve0
prc
pricedelay
rdq
indmom
sgr
std_dolvol
cash
yr
0.000
0.005
0.010
0.015
0.020
0.025
0.030


NN1
yr
month
mom1m
indmom
betasq
idiovol
Year
mve0
baspread
fyear
0.000
0.002
0.004
0.006
0.008
0.010
0.012


NN2
yr
month
indmom
fyear
Year
mve0
sfe
betasq
mom6m
mom1m
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175


NN3
yr
month
Year
betasq
DATE
cash
bm
rd_mve
beta
ep
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014


NN4
yr
month
idiovol
Year
fyear
mve0
mom1m
betasq
beta
time_2
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014


NN5
yr
month
Year
fyear
idiovol
DATE
mve0
mom1m
time_2
:hcapx_ia
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175

We will explore methods to evaluate feature importance across different models. In linear models like Simple Linear Regression and ElasticNet, feature importance is based on the absolute values of the regression coefficients. For the Generalized Linear Model (with Splines & Group Lasso), Group Lasso regularization determines feature importance by the absolute values of the coefficients, grouping related features for flexibility.

Random Forests calculate feature importance by measuring the reduction in Gini impurity during node splits. Features that reduce impurity more significantly are deemed more important.

In GBDT, feature importance is based on each feature's contribution to reducing prediction error. Features that reduce loss during splits are more important, with importance scores provided by `feature_importances_`.

LightGBM measures feature importance using Gain-Based Feature Importance, which evaluates each feature's contribution to reducing the loss function by calculating the cumulative gain across all splits. The importance is determined by the "gain" each feature provides, reflecting its impact on model performance.

Neural Networks use SHAP to determine feature importance. SHAP calculates each feature's marginal contribution to the predicted value, treating each feature as a game participant. In this study, we used part of the training data as a reference to reduce computational costs. The absolute SHAP value reflects feature importance, while the sign indicates whether the feature has a positive or negative impact on the prediction. SHAP is based on the Shapley value from game theory (Aas, K., Jullum, M., & Lland, A. (2021)).

The Shapley value is given by:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|! \cdot (n - |S| - 1)!}{n!} [f(S \cup \{i\}) - f(S)]$$

where S is a subset of features excluding i, N is the set of all features, $f(S)$ is the model's prediction for S, and ϕ_i is the Shapley value of feature i. SHAP optimizes this using a kernel interpreter for high-dimensional data.

5 Difficulties and Future Work

Several challenges arose during the course of our analysis, particularly regarding data completeness and model performance. One of the most significant difficulties was handling the large proportion of missing data, which accounted for approximately 40% of the dataset. To ensure the integrity of the data and the robustness of our empirical analysis, we employed various imputation methods. Following the approach outlined by Dacheng et al. (2020), we filled the missing values using group means for numerical variables such as `betasq`, `chmom`, `dolvol`, `idiovol`, `mve1`, `bm`, and `RET`, calculated at the monthly level. For variables with skewed distributions, like `age`, we used median imputation to mitigate the impact of outliers. Categorical variables, including `sic2` and `datadate`, were filled with the mode, while time-series variables like `fyear` were forward-filled to preserve the temporal structure. Rows with missing crucial identifiers like `gvkey` were excluded, and for highly volatile variables like `std_dolvol` and `std_turn`, standardization was applied. Momentum variables were imputed with the mean of the valid observations within the same month to ensure consistency. These preprocessing techniques ensured the dataset was complete and minimized potential biases from missing data.

In terms of model performance, we encountered significant challenges with the Regression-based model (Generalized Linear Model, GLM), which initially yielded a negative R-squared value, indicating poor fit. To address this, we enhanced the model by incorporating Splines to better capture non-linear relationships and applied Group Lasso regularization to handle multicollinearity and select relevant features. These improvements led to a modest increase in R-squared, suggesting a better representation of the data's underlying patterns. However, further work is needed to refine the model's fit, particularly regarding more complex and non-linear interactions.

Another challenge arose when applying the Gradient Boosting Decision Tree (GBDT), where we observed overfitting, as evidenced by the significant performance gap between the training set ($R^2 = 0.7778$) and the test set ($R^2 = 0.4618$). This was largely due to overfitting resulting from the choice of hyperparameters (learning_rate=0.2, max_depth=5, and n_estimators=200). Additionally, GBDT's sen-

sitivity to outliers was evident, as extreme values led to substantial errors in the test set. Computational efficiency was also a concern, given the large number of trees and the extensive hyperparameter tuning required.

To address these issues, future work will focus on reducing overfitting by lowering the learning rate (e.g., learning_rate=0.05), limiting tree depth (e.g., max_depth=3), and introducing regularization techniques such as L1/L2 penalties and early stopping. Methods to better handle outliers, such as robust scaling and the removal of extreme values, will also be explored. Moreover, experimenting with optimized GBDT variants such as XGBoost or LightGBM will improve computational efficiency and scalability. Expanding the dataset to include more diverse samples will enhance the model's robustness and ability to generalize to new data.

For Neural Networks, we faced challenges in determining optimal model parameters and interpreting the decision-making process. The feature importance analysis was particularly slow, requiring additional computation using SHAP values, complicating the overall interpretation. In the future, we plan to investigate ways to accelerate feature importance calculations and improve model interpretability through alternative methods such as LIME or attention mechanisms.

In conclusion, while our models have provided valuable insights, challenges persist in terms of data preprocessing, model performance, and computational efficiency. Future efforts will be focused on refining the models, particularly in addressing overfitting and improving the handling of missing data. We are also interested in exploring the potential impact of external economic events, such as the 2008 financial crisis, which could provide new directions for future research.

6 Conclusion

This study investigated the application of machine learning techniques to predict US stock returns, comparing the performance of various linear and non-linear models. We employed a comprehensive dataset spanning January 1980 to December 2014, encompassing a wide array of firm characteristics related to valuation, profitability, growth, market dynamics, and trading activity. Our analysis included several regression models (Ordinary Least Squares, ElasticNet, Generalized Linear Model with splines and Group Lasso regularization, Principal Component Regression, and Partial Least Squares), tree-based methods (Random Forest and Gradient Boosting Decision Tree), and neural networks with varying numbers of hidden layers.

Our findings reveal that while linear models, including those incorporating dimensionality reduction and regularization techniques, exhibited limited predictive power (R^2 values consistently below 0.05), non-linear models significantly improved forecast accuracy. Specifically, the LightGBM model demonstrated superior performance, achieving an out-of-sample R^2 of 0.8526, surpassing the accuracy of both Random Forest and GBDT. Interestingly, while increasing the max_depth parameter in tree-based models improved training set performance, it led to overfitting, highlighting the importance of careful hyperparameter tuning. Neural network models showed some volatility in performance depending on the number of hidden layers and optimization algorithm (SGD vs. ADAM), underscoring the need for rigorous model selection.

Feature importance analysis across all models consistently identified three key categories of predictive variables: price trends (momentum at various horizons), volatility (idiosyncratic volatility, beta squared, market beta), and liquidity (bid-ask spread, dollar volume). These variables were consistently ranked among the most influential predictors across a significant portion of our models.

Despite achieving substantial improvements in predictive accuracy with non-linear methods, we encountered challenges in handling missing data (approximately 40% of the dataset) and managing overfitting in certain models. Future research should focus on refining data imputation strategies, further exploring hyperparameter optimization techniques to mitigate overfitting, and potentially incorporating external economic factors to enhance model robustness and explanatory power. The successful application of machine learning techniques to this traditional finance problem opens promising avenues for future research, leveraging the power of advanced algorithms and large datasets to gain deeper insights into financial markets.

References

[1] Aas, K., Jullum, M., & Lland, A. (2021). Explaining individual predictions when features are dependent: more accurate approximations to shapley values. *Artificial Intelligence*. https://doi.org/10.1016/j.artint.2021.103502

[2] Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics, 31*(3), 307-327. Retrieved from https://public.econ.duke.edu/~boller/Published_Papers/joe_86.pdf

[3] Breiman, L. (2001). Random forests. *Machine Learning, 45*(1), 5-32. Retrieved from https://ui.adsabs.harvard.edu/abs/2001MachL..45....5B/abstract

[4] Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785-794. Retrieved from https://www.kdd.org/kdd2016/papers/files/rfp0697-chenAemb.pdf

[5] Engle, R. F. (2001). GARCH 101: The use of ARCH/GARCH models in applied econometrics. *Journal of Economic Perspectives, 15*(4), 157-168. DOI: 10.1257/jep.15.4.157. Retrieved from https://www.aeaweb.org/articles?id=10.1257/jep.15.4.157

[6] Fischer, T., & Krauss, C. (2018). Deep learning with long short-term memory networks for financial market prediction. *European Journal of Operational Research, 270*(2), 654-669. https://doi.org/10.1016/j.ejor.2017.11.054

[7] Gu, S., Kelly, B., & Xiu, D. (2020). Empirical asset pricing via machine learning. *Review of Financial Studies, 33*(5), 2019-2063. Retrieved from https://dachxiu.chicagobooth.edu/download/ML.pdf

[8] He, Y., & Liao, S. (2021). Comparing machine learning models with traditional models for financial data prediction. *Journal of Financial Data Science, 6*(3), 61-79. Retrieved from https://drpress.org/ojs/index.php/cpl/article/download/17363/16850/18892

[9] Jolliffe, I. T. (2002). *Principal Component Analysis* (2nd ed.). Springer-Verlag. Retrieved from http://cda.psych.uiuc.edu/statistical_learning_course/Jolliffe%20I.%20Principal%20Component%20Analysis%20(2ed.,%20Springer,%202002)(518s)_MVsa_.pdf

[10] Kim, H. Y. (2003). The application of support vector machines in financial market prediction. *Expert Systems with Applications, 24*(1), 39-48. DOI: http://dx.doi.org/10.1016/S0305-0483(01)00026-3

[11] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., & Liu, T. Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, 30. Retrieved from https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html

[12] Luwe, Y., Lee, C., & Lim, K. (2023). Wearable sensor-based human activity recognition with ensemble learning: A comparison study. *International Journal of Electrical and Computer Engineering (IJECE)*, *13*(4), 4029–4040. http://doi.org/10.11591/ijece.v13i4.pp4029-4040

[13] Masters, T. (1993). *Practical neural network recipes in C++*. New York: Academic Press. Retrieved from https://www.scirp.org/reference/referencespapers?referenceid=2273637

[14] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological), 58*(1), 267-288. Retrieved from https://webdoc.agsci.colostate.edu/koontz/arec-econ535/papers/Tibshirani%20(JRSS-B%201996).pdf

[15] Timmermann, A. (2018). Forecasting methods in finance. UC San Diego, Rady School of Management. Retrieved from https://rady.ucsd.edu/_files/faculty-research/timmermann/forecasting-methods-in-finance.pdf

[16] Wang, Z. (2020). Deep learning recommendation systems. *Self-Published*, February 2020.

[17] Yan, Z., Li, L., Cheng, L., Chen, X., & Wu, K. (2022). New insight in predicting martensite start temperature in steels. *Journal of Materials Science*, *57*, 1–. https://doi.org/10.1007/s10853-022-07329-y

[18] Zhang, C., Zhang, Y., Cucuringu, M., & Qian, Z. (2023). Volatility forecasting with machine learning and intraday commonality. *Journal of Financial Econometrics*. Retrieved from https://www.researchgate.net/publication/369412581_Volatility_Forecasting_with_Machine_Learning_and_Intraday_Commonality

[19] Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology), 67*(2), 301-320. Retrieved from https://sites.stat.washington.edu/courses/stat527/s13/readings/zouhastie05.pdf

Appendix A: Code for Split Data

```python
import pandas as pd
# 1. Optimize data types
def optimize_data_types(chunk):
# Convert data columns to smaller memory types
chunk['mvel1'] = chunk['mvel1'].astype('float32')  # Convert float64 to float32
chunk['RET'] = chunk['RET'].astype('float32')
chunk['prc'] = chunk['prc'].astype('float32')
chunk['SHROUT'] = chunk['SHROUT'].astype('float32')
chunk['count'] = chunk['count'].astype('int32')  # Convert integers to int32
chunk['permno'] = chunk['permno'].astype('category')  # Convert categorical data to category type
chunk['gvkey'] = chunk['gvkey'].astype('category')  # Convert categorical data to category type
return chunk

# 2. Function to split data by year
def split_data_by_year(chunk, train_start, train_end, val_start, val_end, test_start, test_end):
# Ensure the 'Year' column exists
chunk['Year'] = chunk['DATE'].dt.year
# Filter the training, validation, and test data
train_data = chunk[(chunk['Year'] >= train_start) & (chunk['Year'] <= train_end)]
val_data = chunk[(chunk['Year'] >= val_start) & (chunk['Year'] <= val_end)]
test_data = chunk[(chunk['Year'] >= test_start) & (chunk['Year'] <= test_end)]
return train_data, val_data, test_data

# 3. Rolling window split function
def rolling_window_split(data, start_year, end_year, train_years=18, val_years=12):
# Initialize CSV file paths
train_file = 'train_data.csv'
val_file = 'val_data.csv'
test_file = 'test_data.csv'
first_chunk = True  # Used to control header writing
for year in range(start_year + train_years, end_year):
# Training set: last 18 years
train_start = year - train_years
train_end = year - 1
# Validation set: last 12 years
val_start = year
val_end = year + val_years - 1
# Test set: fixed (1987-2020)
test_start = 1987
test_end = 2020

# Split the data
train_data, val_data, test_data = split_data_by_year(data, train_start, train_end, val_start, val_end, test_start, test_end)

# Save updated dataset once every year
# Write to CSV in chunks, write header only for the first chunk
if first_chunk:
train_data.to_csv(train_file, mode='w', index=False)  # Write header
val_data.to_csv(val_file, mode='w', index=False)  # Write header
test_data.to_csv(test_file, mode='w', index=False)  # Write header
first_chunk = False  # No header for subsequent chunks
else:
train_data.to_csv(train_file, mode='a', header=False, index=False)  # Append data, no header
```

```
val_data.to_csv(val_file, mode='a', header=False, index=False)  # Append data, no header
test_data.to_csv(test_file, mode='a', header=False, index=False)  # Append data, no header
print(f"Completed processing for year {year}")

# Load data and apply rolling window split
chunk_size = 100000  # Number of rows to load at a time
for chunk in pd.read_csv('cleaned_data1.csv', chunksize=chunk_size, parse_dates=['DATE']):
chunk['DATE'] = pd.to_datetime(chunk['DATE'], format='%Y%m%d')  # Force conversion to date format
# Optimize data types
optimized_chunk = optimize_data_types(chunk)

# Perform rolling window split year by year
rolling_window_split(optimized_chunk, start_year=1957, end_year=2020)

print("Data processing and saving completed.")
```

Appendix B: Code for Filling Missing Data

```
import pandas as pd
import numpy as np
# 1. Fill missing values with the group median, using 0 for groups that are all NaN
def fill_missing_with_group_median(data, group_cols, columns_to_fill):
for column in columns_to_fill:
# Use groupby on group_cols and fill missing values with the median of each group
data[column] = data.groupby(group_cols)[column].transform(
lambda x: x.fillna(x.dropna().median() if not x.dropna().empty else 0)  # Fill NaN groups with 0
)
return data

data = pd.read_csv('GHZ_ZHY_V8.csv')  # If the data is stored in a CSV file
data.head()

# 1. Define columns to fill with mean values
columns_to_fill_mean = [
'betasq', 'chmom', 'dolvol', 'idiovol', 'mve1', 'count', 'bm', 'prc', 'ret',
'turn', 'mvel1', 'retvol', 'mve0', 'mom1m', 'mom6m', 'mom12m', 'mom36m',
]
# 2. Define columns to fill with median values
columns_to_fill_median = ['age']
# 3. Define columns to fill with mode values
columns_to_fill_mode = ['sic2', 'STATPERS', 'nanalyst']
# 4. Define a function to drop rows where 'gvkey' is missing
def drop_gvkey_missing(data):
"""Drop rows where 'gvkey' is missing"""
if 'gvkey' in data.columns:
print("Dropping rows where 'gvkey' is missing")
data = data.dropna(subset=['gvkey'])
return data
# 5. Define a function to fill missing values and check the results
def fill_and_check(data, column, fill_method, groupby_columns=['permno', 'yr', 'month']):
"""
Fill missing values for the specified column and check the data after filling.

```

```
:param data: DataFrame
:param column: The column to fill
:param fill_method: The method for filling ('mean', 'median', etc.)
:param groupby_columns: Columns for grouping, default is ['permno', 'yr', 'month']
:return: The DataFrame after filling the missing values
"""
print(f"Filling column: {column} using method: {fill_method}")
if fill_method == 'mean':
# Group by and fill with mean
data[column] = data.groupby(groupby_columns)[column].transform(
lambda x: x.fillna(x.mean()) if not x.isnull().all() else x.fillna(0)
)
elif fill_method == 'median':
# Group by and fill with median
data[column] = data.groupby(groupby_columns)[column].transform(
lambda x: x.fillna(x.median()) if not x.isnull().all() else x.fillna(0)
)
elif fill_method == 'mode':
# Compute mode and fill
mode_value = data[column].mode()[0]
data[column] = data[column].fillna(mode_value)

# Check the data state after filling
check_data(data, f"After Filling {column} with {fill_method.capitalize()}")
return data

# 6. Define a function to fill all features step by step
def fill_all_features(data):
# Drop rows where 'RET' has missing values
data = data.dropna(subset=['RET'])
check_data(data, "After Dropping RET Missing")

# Fill features one by one
for col in columns_to_fill_mean:
if col in data.columns:
data = fill_and_check(data, col, 'mean')

for col in columns_to_fill_median:
if col in data.columns:
data = fill_and_check(data, col, 'median')

for col in columns_to_fill_mode:
if col in data.columns:
data = fill_and_check(data, col, 'mode')

# Drop rows where 'gvkey' is missing
data = drop_gvkey_missing(data)
check_data(data, "After Dropping Missing gvkey Rows")

return data
```

Appendix C: Code for Simple Linear Regression and Feature Importance Analysis

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.metrics import r2_score, mean_squared_error

# Step 1: Load data
train_data = pd.read_csv("C:\\train_data.csv")
val_data = pd.read_csv("C:\\val_data.csv")
test_data = pd.read_csv("C:\\test_data.csv")

# Step 2: Assume the target variable is 'RET'
target_variable = 'RET'

# Step 3: Exclude date column and non-numeric columns
X_train = train_data.drop([target_variable, 'DATE'], axis=1)
X_val = val_data.drop([target_variable, 'DATE'], axis=1)
X_test = test_data.drop([target_variable, 'DATE'], axis=1)

# Step 4: Ensure all feature columns are numeric
X_train = X_train.select_dtypes(include=['float64', 'int64'])
X_val = X_val.select_dtypes(include=['float64', 'int64'])
X_test = X_test.select_dtypes(include=['float64', 'int64'])

# Step 5: Extract the target variable
y_train = train_data[target_variable]
y_val = val_data[target_variable]
y_test = test_data[target_variable]

# Step 6: Exclude features containing "yr", "Year", and "date"
keywords_to_exclude = ["yr", "year", "date"]
X_train_filtered = X_train.loc[:, ~X_train.columns.str.contains('|'.join(
    keywords_to_exclude), case=False)]
X_val_filtered = X_val.loc[:, ~X_val.columns.str.contains('|'.join(keywords_to_exclude
    ), case=False)]
X_test_filtered = X_test.loc[:, ~X_test.columns.str.contains('|'.join(
    keywords_to_exclude), case=False)]

# Step 7: Feature standardization
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_filtered)
X_val_scaled = scaler.transform(X_val_filtered)
X_test_scaled = scaler.transform(X_test_filtered)

# Step 8: Add a constant term (intercept _0)
X_train_scaled = np.c_[np.ones(X_train_scaled.shape[0]), X_train_scaled]
X_val_scaled = np.c_[np.ones(X_val_scaled.shape[0]), X_val_scaled]
X_test_scaled = np.c_[np.ones(X_test_scaled.shape[0]), X_test_scaled]

# Step 9: Calculate regression coefficients ^()
# ^ = (X^T X)^(-1) X^T y
X_train_transpose = X_train_scaled.T
beta_hat = np.linalg.inv(X_train_transpose.dot(X_train_scaled)).dot(X_train_transpose)
    .dot(y_train)

# Step 10: Predict on training and testing datasets
y_pred_train = X_train_scaled.dot(beta_hat)  # Predictions on training set
y_pred_test = X_test_scaled.dot(beta_hat)    # Predictions on test set

# Step 11: Calculate R² and MSE for training and testing datasets
r2_train = r2_score(y_train, y_pred_train)   # R² for training set
r2_test = r2_score(y_test, y_pred_test)     # R² for test set

# Step 12: Print R²
print(f"Train R²: {r2_train}")
```

```
print(f"Test R²: {r2_test}")

# Step 13: Calculate feature importance and plot the top 10
# Feature importance is based on the absolute value of regression coefficients (excluding intercept)
coefficients = beta_hat[1:]  # Exclude intercept term
features = X_train_filtered.columns  # Feature names
importance = pd.DataFrame({'Feature': features, 'Coefficient': np.abs(coefficients)})

# Step 14: Sort by absolute values and select the top 10 most important features
importance = importance.sort_values(by='Coefficient', ascending=False)
top_10_importance = importance.head(10)

# Step 15: Plot feature importance
plt.figure(figsize=(10, 6))
plt.barh(top_10_importance['Feature'], top_10_importance['Coefficient'], color='#1E344D')
plt.gca().invert_yaxis()  # Invert y-axis to show the most important feature on top
plt.title("OLS")
plt.show()
```

Appendix D: Code for ElasticNet Regression and Feature Importance Analysis

```
import pandas as pd
import numpy as np
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error
import matplotlib.pyplot as plt

# Step 1: Load data
train_data = pd.read_csv("C:\\train_data.csv")
val_data = pd.read_csv("C:\\val_data.csv")
test_data = pd.read_csv("C:\\test_data.csv")

# Step 2: Assume the target variable is 'RET', representing asset returns
target_variable = 'RET'

# Step 3: Exclude non-numeric columns and the 'DATE' column, keeping only numeric features
X_train = train_data.drop([target_variable, 'DATE'], axis=1).select_dtypes(include=['float64', 'int64'])
X_val = val_data.drop([target_variable, 'DATE'], axis=1).select_dtypes(include=['float64', 'int64'])
X_test = test_data.drop([target_variable, 'DATE'], axis=1).select_dtypes(include=['float64', 'int64'])

# Step 4: Exclude features containing "yr", "Year", and "date"
keywords_to_exclude = ["yr", "year", "date"]
X_train_filtered = X_train.loc[:, ~X_train.columns.str.contains('|'.join(keywords_to_exclude), case=False)]
X_val_filtered = X_val.loc[:, ~X_val.columns.str.contains('|'.join(keywords_to_exclude), case=False)]
X_test_filtered = X_test.loc[:, ~X_test.columns.str.contains('|'.join(keywords_to_exclude), case=False)]

```

```
# Step 5: Extract the target variable (asset returns)
y_train = train_data[target_variable]
y_val = val_data[target_variable]
y_test = test_data[target_variable]

# Step 6: Data standardization (ElasticNet is sensitive to feature scales, so scaling is necessary)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_filtered)  # Standardize training set
X_val_scaled = scaler.transform(X_val_filtered)          # Standardize validation set
X_test_scaled = scaler.transform(X_test_filtered)        # Standardize test set

# Step 7: Define the ElasticNet model
elasticnet = ElasticNet()

# Step 8: Define hyperparameter search space
param_grid = {
        'alpha': [0.01, 0.1, 0.5, 1.0, 10.0, 100.0],  # Regularization strength (controls overall regularization effect)
        'l1_ratio': [0.1, 0.3, 0.5, 0.7, 0.9, 1.0]   # Mix ratio of L1 and L2 (0 = pure L2, 1 = pure L1)
}

# Step 9: Perform hyperparameter tuning with GridSearchCV, using R² as the evaluation metric
grid_search = GridSearchCV(estimator=elasticnet, param_grid=param_grid, scoring='r2', cv=5)
grid_search.fit(X_train_scaled, y_train)  # Train the model and find the best hyperparameters

# Step 10: Print the best parameters
print("Best parameters found: ", grid_search.best_params_)

# Step 11: Retrain ElasticNet model with the best parameters
best_model = grid_search.best_estimator_

# Step 12: Make predictions on training and test datasets
y_pred_train = best_model.predict(X_train_scaled)  # Predictions on training set
y_pred_test = best_model.predict(X_test_scaled)    # Predictions on test set

# Step 13: Calculate R² and MSE for training and testing datasets
r2_train = r2_score(y_train, y_pred_train)  # R² for training set
r2_test = r2_score(y_test, y_pred_test)     # R² for test set

# Step 14: Print R² for training and testing datasets
print(f"ElasticNet Regression - Train R²: {r2_train}")
print(f"ElasticNet Regression - Test R²: {r2_test}")

# Step 15: Calculate feature importance based on ElasticNet coefficients
coefficients = best_model.coef_  # Retrieve coefficients of the trained ElasticNet model
features = X_train_filtered.columns  # Feature names
importance = pd.DataFrame({'Feature': features, 'Coefficient': np.abs(coefficients)})

# Step 16: Sort by absolute values of the coefficients and select the top 10 features
importance = importance.sort_values(by='Coefficient', ascending=False)
top_10_importance = importance.head(10)

# Step 17: Plot the top 10 feature importances
plt.figure(figsize=(10, 6))
```

```
plt.barh(top_10_importance['Feature'], top_10_importance['Coefficient'], color='#1E344D')
plt.gca().invert_yaxis()  # Invert y-axis to display the most important feature on top
plt.title("ENet")
plt.show()
```

Appendix E: Code for Applying PCA and PLS Regression to Financial Time Series

```
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
train_data = pd.read_csv('train_data.csv')
test_data = pd.read_csv('test_data.csv')
val_data = pd.read_csv('val_data.csv')
# Update the dataset
X_train = train_data.drop(columns=['RET', 'Year', 'month'])
y_train = train_data['RET']
X_test = test_data.drop(columns=['RET', 'Year', 'month'])
y_test = test_data['RET']
X_val = val_data.drop(columns=['RET', 'Year', 'month'])
y_val = val_data['RET']
# 3. Standardize the data (PCR requires feature data to be standardized)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
X_val_scaled = scaler.transform(X_val)
# PCA dimensionality reduction
pca = PCA(n_components=0.85)  # Select the principal components that explain 85% of the variance
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)
X_val_pca = pca.transform(X_val_scaled)
# Output the number of selected principal components
print("Number of selected principal components:", pca.n_components_)
# PLS
# Initialize the PLS model, assuming 5 principal components are selected
pls = PLSRegression(n_components=5)
# Train the model
pls.fit(X_train_scaled, y_train)
# Predict on the test and validation sets
y_pred_test = pls.predict(X_test_scaled)
y_pred_val = pls.predict(X_val_scaled)
# Calculate model performance
mse_test = mean_squared_error(y_test, y_pred_test)
mse_val = mean_squared_error(y_val, y_pred_val)
r2_test = r2_score(y_test, y_pred_test)
r2_val = r2_score(y_val, y_pred_val)
print(f"PLS Test MSE: {mse_test}")
print(f"PLS Validation MSE: {mse_val}")
print(f"PLS Test R²: {r2_test}")
print(f"PLS Validation R²: {r2_val}")
```

Appendix F: Code for Generalized Linear Regression and Feature Importance Analysis

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
from sklearn.metrics import r2_score
from patsy import dmatrix  # Used to generate spline basis functions
import matplotlib.pyplot as plt
from group_lasso import GroupLasso

# Step 1: Load data
train_data = pd.read_csv("C:\\train_data.csv")
val_data = pd.read_csv("C:\\val_data.csv")
test_data = pd.read_csv("C:\\test_data.csv")

# Step 2: Assume the target variable is 'RET'
target_variable = 'RET'

# Step 3: Exclude the date column and non-numeric columns
X_train = train_data.drop([target_variable, 'DATE'], axis=1)
X_val = val_data.drop([target_variable, 'DATE'], axis=1)
X_test = test_data.drop([target_variable, 'DATE'], axis=1)

# Step 4: Exclude features containing "yr", "Year", and "date"
keywords_to_exclude = ["yr", "year", "date"]
X_train_filtered = X_train.loc[:, ~X_train.columns.str.contains('|'.join(
    keywords_to_exclude), case=False)]
X_val_filtered = X_val.loc[:, ~X_val.columns.str.contains('|'.join(keywords_to_exclude
    ), case=False)]
X_test_filtered = X_test.loc[:, ~X_test.columns.str.contains('|'.join(
    keywords_to_exclude), case=False)]

# Step 5: Ensure all feature columns are numeric
X_train_filtered = X_train_filtered.select_dtypes(include=['float64', 'int64'])
X_val_filtered = X_val_filtered.select_dtypes(include=['float64', 'int64'])
X_test_filtered = X_test_filtered.select_dtypes(include=['float64', 'int64'])

# Step 6: Extract the target variable
y_train = train_data[target_variable]
y_val = val_data[target_variable]
y_test = test_data[target_variable]

# Step 7: Spline regression: Generate spline basis functions for each feature
spline_features_train = []
spline_features_val = []
spline_features_test = []

for col in X_train_filtered.columns:
# Use natural splines to generate basis functions
spline_train = dmatrix("bs(x, df=5, include_intercept=False)", {"x": X_train_filtered[
    col]}, return_type='dataframe')
spline_val = dmatrix("bs(x, df=5, include_intercept=False)", {"x": X_val_filtered[col
    ]}, return_type='dataframe')
spline_test = dmatrix("bs(x, df=5, include_intercept=False)", {"x": X_test_filtered[
    col]}, return_type='dataframe')

spline_features_train.append(spline_train)
spline_features_val.append(spline_val)
spline_features_test.append(spline_test)
```

```
52
53 # Step 8: Combine spline features into a single DataFrame
54 X_train_spline = pd.concat(spline_features_train, axis=1)
55 X_val_spline = pd.concat(spline_features_val, axis=1)
56 X_test_spline = pd.concat(spline_features_test, axis=1)
57
58 # Step 9: Add a constant term
59 X_train_spline = sm.add_constant(X_train_spline)
60 X_val_spline = sm.add_constant(X_val_spline)
61 X_test_spline = sm.add_constant(X_test_spline)
62
63 # Step 10: Use Group Lasso for regularization
64 group_lasso = GroupLasso(alpha=0.1, groups=[list(range(i, i+5)) for i in range(0, len(
       X_train_spline.columns), 5)])  # Every 5 features form a group
65 group_lasso.fit(X_train_spline, y_train)
66
67 # Step 11: View the regularized coefficients
68 print(group_lasso.coef_)
69
70 # Step 12: Make predictions on training and test datasets
71 y_pred_train_glm = group_lasso.predict(X_train_spline)
72 y_pred_test_glm = group_lasso.predict(X_test_spline)
73
74 # Step 13: Calculate R² for training and testing datasets
75 r2_train_glm = r2_score(y_train, y_pred_train_glm)
76 r2_test_glm = r2_score(y_test, y_pred_test_glm)
77
78 print(f"Generalized Linear Model with Splines and Group Lasso - Train R²: {
       r2_train_glm}")
79 print(f"Generalized Linear Model with Splines and Group Lasso - Test R²: {r2_test_glm}
       ")
80
81 # Step 14: Calculate feature importance based on absolute values of the coefficients
82 coefficients = group_lasso.coef_[1:]  # Exclude the intercept term
83 importance = pd.DataFrame({
84         'Feature': X_train_spline.columns[1:],  # Exclude the intercept
85         'Coefficient': np.abs(coefficients)
86 })
87
88 # Step 15: Sort the coefficients by absolute value
89 importance = importance.sort_values(by='Coefficient', ascending=False)
90
91 # Step 16: Select the top 10 features
92 top_10_importance = importance.head(10)
93
94 # Step 17: Plot the top 10 feature importances
95 plt.figure(figsize=(10, 6))
96 plt.barh(top_10_importance['Feature'], top_10_importance['Coefficient'], color='#1
       E344D')
97 plt.gca().invert_yaxis()  # Display the most important feature on top
98 plt.title("GLM")
99 plt.show()
```

Appendix G: Code for Random Forest Regressor Model Training, Hyperparameter Tuning, and Performance Evaluation on Financial Data

```
1 import pandas as pd
2 import numpy as np
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split

# Step 1: Read data
# Load training, testing, and validation datasets from CSV files
train_data = pd.read_csv('train_data.csv')
test_data = pd.read_csv('test_data.csv')
val_data = pd.read_csv('val_data.csv')

# Step 2: Select Features
# Define the list of features (independent variables) to be used in the model
#features = [
#    'prc', 'prc_ma3', 'momentum_5', 'RET_lag1', 'mom6m', 'chpmia', 'cashpr', 'mom12m', #'sgr', 'baspread',
#    'indmom', 'retvol', 'mvel1', 'mom1m', 'permno', 'DATE', 'SHROUT', 'count', 'beta', #'betasq', 'chmom',
#    'chnanalyst', 'dolvol', 'idiovol', 'IPO', 'mom1m', 'mom6m', 'mom12m', 'mom36m', #'mve0', 'pricedelay',
#    'turn', 'time_1', 'SHRCD', 'EXCHCD', 'time_2', 'gvkey', 'fyear', 'absacc', 'acc', 'age', 'agr', # 'bm',
#    'bm_ia', 'cashdebt', 'cfp', 'cfp_ia', 'chatoia', 'chcsho', 'chempia', 'chinv', 'convind', 'currat', 'depr',
#    'divi', 'divo', 'dy', 'egr', 'ep', 'gma', 'grcapx', 'grltnoa', 'herf', 'hire', 'invest', 'lev', 'lgr', #'mve_ia',
#    'operprof', 'mve_f', 'orgcap', 'pchcapx_ia', 'pchcurrat', 'pchdepr', 'pchgm_pchsale', 'pchquick',
#    'pchsale_pchinvt', 'pchsale_pchrect', 'pchsale_pchxsga', 'pchsaleinv', 'pctacc', 'ps', #'quick', 'rd',
#    'rd_mve', 'rd_sale', 'realestate', 'roic', 'salecash', 'saleinv', 'salerec', 'secured', #'securedind', 'sin',
#    'sp', 'tang', 'tb', 'datadate', 'aeavol', 'cash', 'chtx', 'cinvest', 'ear', 'nincr', 'roaq', 'roavol', #'roeq',
#    'rsup', 'stdacc', 'stdcf', 'sue', 'rdq', 'ms', 'chfeps', 'disp', 'fgr5yr', 'nanalyst', 'sfe', 'yr', #'month',
#    'ill', 'maxret', 'retvol', 'std_dolvol', 'std_turn', 'zerotrade', 'sic2'
#]
#features = ['prc', 'prc_ma3', 'momentum_5', 'RET_lag1', 'mom6m', 'chpmia', 'cashpr', 'mom12m', 'sgr', 'baspread', 'indmom', 'retvol', 'mvel1', 'mom1m']
#features = ['prc', 'prc_ma3', 'momentum_5', 'RET_lag1', 'mom6m', 'chpmia', 'cashpr', #'mom12m', 'sgr', 'baspread', 'indmom', 'retvol', 'mvel1', 'mom1m']
features = ['prc', 'prc_ma3', 'momentum_5', 'RET_lag1', 'mom6m']

# Split the datasets into independent variables (X) and target variable (y)
X_train, y_train = train_data[features], train_data['RET']
X_test, y_test = test_data[features], test_data['RET']
X_val, y_val = val_data[features], val_data['RET']

# Step 3: Feature Engineering
# Create additional features based on rolling windows and lag features

# Calculate the 3-period moving average of 'prc' for each 'permno' (unique identifier)
train_data['prc_ma3'] = train_data.groupby('permno')['prc'].transform(lambda x: x.rolling(window=3, min_periods=1).mean())
train_data['retvol_ma10'] = train_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=10, min_periods=1).std())
```

```
train_data['momentum_5'] = train_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=5, min_periods=1).sum())
train_data['momentum_10'] = train_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=10, min_periods=1).sum())
train_data['volume_change'] = train_data.groupby('permno')['dolvol'].transform(lambda x: x.pct_change().fillna(0))
train_data['RET_lag1'] = train_data.groupby('permno')['RET'].shift(1).fillna(0)
train_data['RET_lag5'] = train_data.groupby('permno')['RET'].shift(5).fillna(0)

# Apply the same transformations to the test and validation datasets
test_data['prc_ma3'] = test_data.groupby('permno')['prc'].transform(lambda x: x.rolling(window=3, min_periods=1).mean())
test_data['retvol_ma10'] = test_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=10, min_periods=1).std())
test_data['momentum_5'] = test_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=5, min_periods=1).sum())
test_data['momentum_10'] = test_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=10, min_periods=1).sum())
test_data['volume_change'] = test_data.groupby('permno')['dolvol'].transform(lambda x: x.pct_change().fillna(0))
test_data['RET_lag1'] = test_data.groupby('permno')['RET'].shift(1).fillna(0)
test_data['RET_lag5'] = test_data.groupby('permno')['RET'].shift(5).fillna(0)

val_data['prc_ma3'] = val_data.groupby('permno')['prc'].transform(lambda x: x.rolling(window=3, min_periods=1).mean())
val_data['retvol_ma10'] = val_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=10, min_periods=1).std())
val_data['momentum_5'] = val_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=5, min_periods=1).sum())
val_data['momentum_10'] = val_data.groupby('permno')['RET'].transform(lambda x: x.rolling(window=10, min_periods=1).sum())
val_data['volume_change'] = val_data.groupby('permno')['dolvol'].transform(lambda x: x.pct_change().fillna(0))
val_data['RET_lag1'] = val_data.groupby('permno')['RET'].shift(1).fillna(0)
val_data['RET_lag5'] = val_data.groupby('permno')['RET'].shift(5).fillna(0)

# Step 4: Data Preprocessing
# Handle missing values using the median strategy for imputation
imputer = SimpleImputer(strategy='median')
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)
X_val_imputed = imputer.transform(X_val)

# Step 5: Feature Scaling
# Standardize the features to have zero mean and unit variance
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_imputed)
X_test_scaled = scaler.transform(X_test_imputed)
X_val_scaled = scaler.transform(X_val_imputed)

# Step 6: Model Training
# Initialize the RandomForestRegressor model and train it on the scaled training data
model = RandomForestRegressor(
n_estimators=350,
max_depth=10,
min_samples_split=5,
min_samples_leaf=10,
max_features='sqrt',
random_state=42,
n_jobs=-1
)
```

```
model.fit(X_train_scaled, y_train)

# Step 7: Model Evaluation
# Predict on the test data and evaluate the model's performance using different metrics
y_pred_test = model.predict(X_test_scaled)
mse_test = mean_squared_error(y_test, y_pred_test)
r2_test = r2_score(y_test, y_pred_test)
mae_test = np.mean(np.abs(y_test - y_pred_test))
mape_test = np.mean(np.abs((y_test - y_pred_test) / np.clip(y_test, 1e-8, None))) * 100

# Print evaluation metrics
print(f"Test Set - Mean Squared Error (MSE): {mse_test}")
print(f"Test Set - R-squared (R²): {r2_test}")
print(f"Test Set - Mean Absolute Error (MAE): {mae_test}")
print(f"Test Set - Mean Absolute Percentage Error (MAPE): {mape_test}%")

# Step 8: Feature Importance
# Get the feature importances from the trained model and display the top 10 most important features
feature_importances = model.feature_importances_
feature_importance_df = pd.DataFrame({
        'Feature': features,
        'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

top_features = feature_importance
```

Appendix H: Code for GBDT Feature Importance Extraction

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import GradientBoostingRegressor

# Extract features and target variable
X = df.iloc[:, :-1]
y = df.iloc[:, -1]

# Normalize the features
scaler = MinMaxScaler()
X_normalized = scaler.fit_transform(X)

# Convert normalized features to DataFrame
X_normalized_df = pd.DataFrame(X_normalized, columns=X.columns)

# Fit GBDT model to calculate feature importance
model = GradientBoostingRegressor()
model.fit(X_normalized_df, y)
feature_importances = model.feature_importances_

# Create a DataFrame with feature importance scores
feature_scores = pd.Series(feature_importances, index=X.columns).sort_values(ascending=False)

# Print feature importance scores
print("Feature Importances:")
print(feature_scores)

```

```
# 1. Output feature importance to Excel
feature_scores.to_excel('feature_importances_GBDT.xlsx', sheet_name='Feature Importance')

# 2. Select the top 10 most important features
top_10_features = feature_scores.head(10).index

# Create a DataFrame with the top 10 features and target variable
top_10_df = df[top_10_features.tolist() + [y.name]]

# Output the top 10 features to a new Excel file
top_10_df.to_excel('top_10_features_GBDT.xlsx', index=False)
```

Appendix I: Code for Gradient Boosting Decision Tree Hyperparameter Tuning and Performance Evaluation

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Step 1: Parameter Grid
param_grid = {
        'n_estimators': [50, 100, 200],
        'learning_rate': [0.01, 0.1, 0.2],
        'max_depth': [3, 4, 5],
}

# Step 2: Initialize GBDT
gbdt = GradientBoostingRegressor(random_state=42)

# Step 3: Grid Search
grid_search = GridSearchCV(estimator=gbdt, param_grid=param_grid, cv=5, n_jobs=-1, verbose=2)
grid_search.fit(X, y)

# Step 4: Best Parameters
best_params = grid_search.best_params_
print("Best parameter combination:", best_params)

# Step 5: Retrain
best_gbdt = grid_search.best_estimator_
best_gbdt.fit(X, y)

# Step 6: Predict
y_pred = best_gbdt.predict(X)

# Step 7: Evaluate
mse = mean_squared_error(y, y_pred)
mae = mean_absolute_error(y, y_pred)
r2 = r2_score(y, y_pred)

# Step 8: Results
print(f"MAE: {mae:.4f}")
print(f"MSE: {mse:.4f}")
print(f"R²: {r2:.4f}")
```

Appendix J: Code for LightGBM Hyperparameter Tuning and Feature Importance Analysis

```
import pandas as pd
import lightgbm as lgb
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt

# Step 1: Load training and testing datasets
train_data = pd.read_csv('train_data.csv')
test_data = pd.read_csv('test_data.csv')

# Step 2: Separate features (X) and target variable (y)
X_train = train_data.drop(columns=['RET'])
y_train = train_data['RET']
X_test = test_data.drop(columns=['RET'])
y_test = test_data['RET']

# Step 3: Drop unnecessary columns
columns_to_drop = ['Year', 'month', 'DATE', 'STATPERS']
X_train.drop(columns=columns_to_drop, inplace=True, errors='ignore')
X_test.drop(columns=columns_to_drop, inplace=True, errors='ignore')

# Step 4: Define maximum depth values for hyperparameter tuning
depths = [5, 6, 7, 8]

# Step 5: Initialize containers for R² results and feature importance
r2_results = []
importance_results = {}

# Step 6: Iterate over each depth and train LightGBM
for depth in depths:
print(f"Training LightGBM with max_depth={depth}")

# Define LightGBM parameters
params = {
        'objective': 'regression',        # Regression task
        'metric': 'rmse',                # Use root mean squared error as the metric
        'learning_rate': 0.015,          # Learning rate
        'max_depth': depth,              # Maximum depth of trees
        'num_leaves': 2**depth - 1,      # Maximum number of leaves per tree
        'verbose': -1                    # Suppress output
}

# Create LightGBM datasets
train_dataset = lgb.Dataset(X_train, label=y_train)
test_dataset = lgb.Dataset(X_test, label=y_test, reference=train_dataset)

# Train the LightGBM model
model = lgb.train(
params,
train_set=train_dataset,
num_boost_round=500  # Number of boosting iterations
)

# Predict on training and testing datasets
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# Calculate R² for training and testing
```

```
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)
r2_results.append({'Depth': depth, 'R² Train': r2_train, 'R² Test': r2_test})

# Calculate feature importance
importance = model.feature_importance(importance_type='gain')
feature_names = X_train.columns
importance_percent = importance / sum(importance) / 10
importance_df = pd.DataFrame({'Feature': feature_names, 'Importance':
    importance_percent})
importance_df = importance_df.sort_values(by='Importance', ascending=False).head(10)

# Plot feature importance for the top 10 features
plt.figure(figsize=(6, 4))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='#1E344D')
plt.title(f'LightGBM(max_depth={depth})')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()

# Store feature importance results
importance_results[depth] = importance_df

# Convert R² results into a DataFrame and display
r2_df = pd.DataFrame(r2_results)
print(r2_df)
```

Appendix K: Code for Neural Network Training, Hyperparameter Tuning, and feature importance Analysis

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import shap
import tensorflow as tf
from sklearn.metrics import accuracy_score
from sklearn.metrics import r2_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization
from tensorflow.keras.regularizers import l1_l2
from tensorflow.keras.optimizers import SGD, Adam

# Step 1: Input data
train_data=pd.read_csv‘(train_data.’csv)
test_data=pd.read_csv‘(test_data.’csv)
val_data=pd.read_csv‘(val_data.’csv)

# Step 2: Select Features
X=train_data1.drop(columns=['RET'])
Y=train_data1['RET']
feature_names=X.columns.tolist()

# Step 3: Build Model
Def build_nn(input_dim,layers,l1_reg=0.0,l2_reg=0.01, learning_rate=0.001,
    optimizer_type='sgd'):
model=Sequential()
# Input layer and first hidden layer
```

```
model.add(Dense(layers[0], input_dim=input_dim, activation='relu', kernel_regularizer=
    l1_l2(l1=l1_reg, l2=l2_reg)))
model.add(BatchNormalization())   # Batch normalization
for units in layers[1:]: # Add other hidden layers
model.add(Dense(units, activation='relu',kernel_regularizer=l1_l2(l1=l1_reg, l2=l2_reg
    )))
model.add(BatchNormalization())
model.add(Dense(1, activation='linear')) # Single output
# Select the optimizer
if optimizer_type == 'adam':
optimizer = Adam(learning_rate=learning_rate)
elif optimizer_type=='sgd':
optimizer = SGD(learning_rate=learning_rate, momentum=0.9, nesterov=True)
model.compile(optimizer=optimizer, loss='mse', metrics=['mae'])
return model
}
# Build an NN5 model with one hidden layer (32,16,8,4,2)
input_dim= X_train.shape[1] # Number of features
layers=[32, 16, 8, 4, 2]
model= build_nn(input_dim, layers, l2_reg=0.01, learning_rate=0.001, optimizer_type='
    sgd')

# Step 4:  Training model
history = model.fit(
X_train, Y_train,
validation_data=(X_val, Y_val),
epochs=40,
batch_size=32,
verbose=1
# Step 5: Test_data evaluation
Y_test_numpy=Y_test.to_numpy()
Y_pre=model.predict(X_test)
test_loss, test_mae = model.evaluate(X_test, Y_test, verbose=0)
print(f"Test Loss (MSE): {test_loss:.4f}, Test MAE: {test_mae:.4f}")
r2_sklearn = r2_score(Y_test, Y_pre)
print(f"R^2 (using sklearn): {r2_sklearn:.4f}")

# Step 6: Feature contribution and visualization
explainer = shap.KernelExplainer(model.predict, X_train[:100])  # Use partial training
     data (100 samples)
shap_values = explainer.shap_values(X_train[:100])  # Calculate the SHAP
    valuemean_importance = np.abs(shap_values).mean(axis=0) # Calculate the average
    feature importance
importance_df = pd.DataFrame({
'Feature': feature_names,
'Importance': mean_importance
})
importance_df = importance_df.sort_values(by='Importance', ascending=False)
top_10_features = importance_df.head(10)
# visualization
plt.figure(figsize=(6, 4))
plt.barh(top_10_features['Feature'], top_10_features['Importance'], color='#1E344D')
#plt.xlabel('Importance')
#plt.ylabel('Features')
plt.title('NN5')
plt.gca().invert_yaxis()
plt.show()
```