FA590 Statistical Learning in Finance

Final Project

Stock-Level Risk Premiums Analysis and Prediction

Instructor: Professor Yang
Semester: Fall 2024

Group Members: De Alarcón Lucía and Malamisura Federica

Contents

Abstract

This report details the implementation of statistical learning models to predict stock-level risk premiums. Leveraging a comprehensive dataset covering 11 years of stock data with firm-specific characteristics and macroeconomic variables, we employ various machine learning algorithms, including linear regression, decision trees, ensemble methods, and deep neural networks. GPU acceleration using CUDA and cuDNN was utilized to enhance computational efficiency. The models are evaluated using metrics such as Mean Squared Error (MSE) and R-squared (R^2) score. The results indicate that advanced ensemble methods and neural networks outperform traditional models in predicting risk premiums, highlighting the potential of machine learning techniques in financial forecasting.

1 Introduction

1.1 Objective and Description of the Project

The primary objective of this project is to design, implement, and evaluate advanced statistical learning models to predict stock-level risk premiums. The risk premium represents the additional return an investor expects to receive for taking on the risk of holding a volatile asset instead of a risk-free asset. Accurate prediction of risk premiums is crucial for optimizing portfolio returns, managing risk exposure, and informing investment strategies. By leveraging modern machine learning techniques, this project aims to improve the precision and reliability of such predictions, addressing the limitations of traditional financial models.

We utilized a dataset spanning 60 years of stock data, comprising approximately 30,000 individual stocks. For each stock, the dataset includes 94 firm-specific characteristics, along with eight aggregate time-series variables that represent macroeconomic conditions. This dataset captures the relationship between company-specific attributes and macroeconomic factors, providing a rich source of information for predicting risk premiums.

The dataset includes the following types of variables:

- **Firm-specific characteristics:** Variables such as profitability, leverage, market capitalization, momentum indicators, and valuation metrics provide insights into individual stock performance and financial health.
- **Macroeconomic indicators:** Variables such as the dividend-price ratio, earnings-price ratio, term spread, and default yield spread reflect the broader economic environment and its influence on asset pricing.

This extensive dataset offers a unique opportunity to combine granular firm-level information with overarching economic trends to build robust predictive models.

1.2 Justification for Choosing a Machine Learning Approach

Traditional financial models, such as linear regression, rely on simplifying assumptions that often fail to capture the complex, nonlinear relationships inherent in financial data. In contrast, machine learning models, including neural networks and gradient-boosting algorithms, excel at uncovering hidden patterns and interactions within large, high-dimensional datasets.

By employing machine learning approaches, we aim to:

- Improve prediction accuracy by modeling nonlinear and high-order interactions between features.
- Handle the high dimensionality of the dataset effectively.
- Adapt to evolving market dynamics by leveraging flexible algorithms.

Additionally, the use of GPU acceleration through CUDA and cuDNN enables the efficient training of computationally intensive models, significantly reducing runtime and allowing for iterative experimentation with larger architectures and datasets.

2 Exploratory Data Analysis

2.1 Summary Statistics

An initial exploration of the dataset revealed several key insights. The target variable, `risk_premium`, has a mean of approximately 0.69 and a standard deviation of 17.49, highlighting substantial variability in stock returns.

2.2 Visualizations

To better understand the data, we performed the following visualizations:

- **Distribution of Risk Premium:** A histogram with a kernel density estimate (Figure 1) illustrates the distribution of the `risk_premium` variable. The plot reveals a slight skew, with several extreme outliers, emphasizing the need for robust models to handle such data.

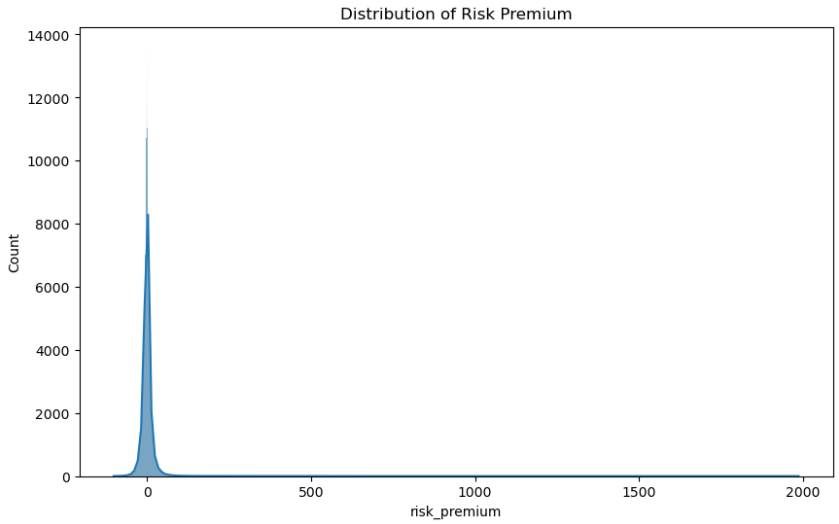


Figure 1: Distribution of Risk Premium

- **Missing Values Matrix:** Using a missingno matrix, we visualized the pattern of missing values in the dataset. Identifying and addressing missing data is critical for ensuring model reliability.
- **Correlation Heatmap:** A heatmap of the feature correlation matrix (Figure 2) was created to detect multicollinearity among features. Highly correlated variables may lead to redundancy and instability in predictive models.

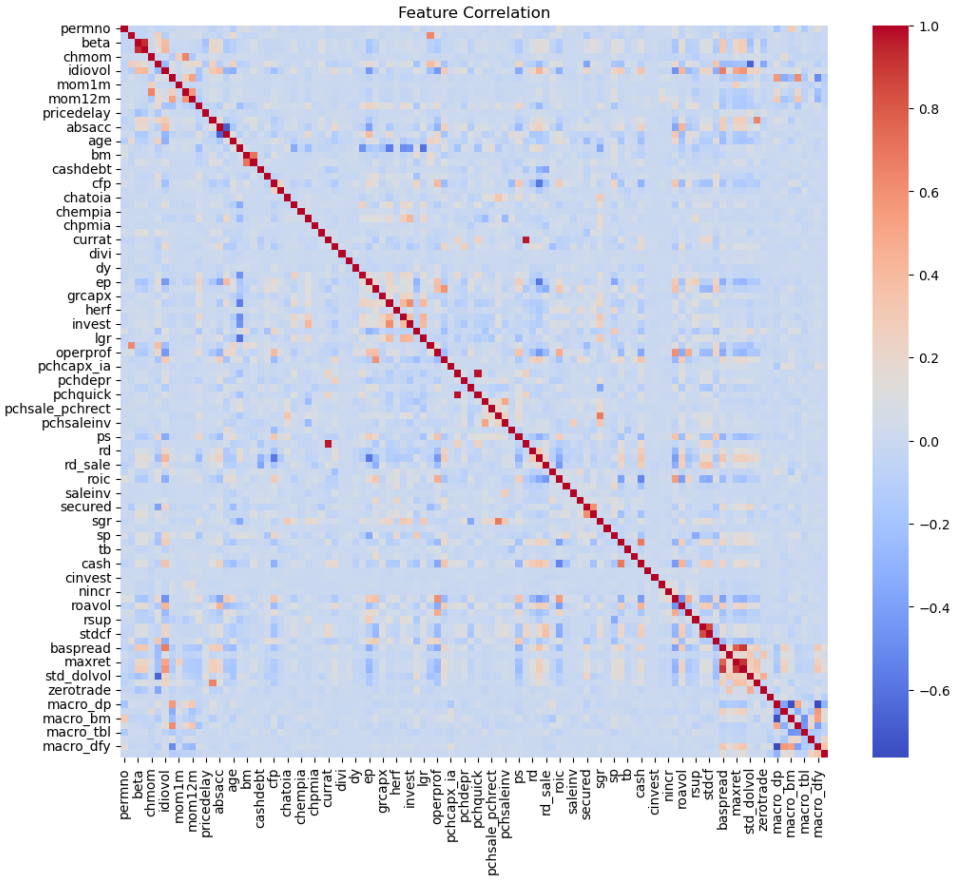


Figure 2: Feature Correlation Heatmap

2.3 Insights

The exploratory analysis provided several insights:

- **Risk Premium Distribution:** The distribution of the `risk_premium` variable is slightly skewed, with a few extreme outliers. These outliers may impact model training, suggesting the potential need for robust regression techniques or outlier handling - which we will discuss later.
- **Multicollinearity:** The correlation heatmap indicates that some features exhibit high levels of multicollinearity, which can adversely affect model performance.
- **Data Gaps:** The missing values visualization underscores the importance of careful data preprocessing. Imputation methods or data exclusion strategies must be employed to ensure consistent and unbiased predictions.

The insights gained from this analysis informed our preprocessing pipeline and model selection process, ensuring that the data was clean, interpretable, and suitable for advanced machine learning techniques.

3 Data Preprocessing

3.1 Handling Missing Values

To ensure data consistency and reliability, we carefully addressed missing values in the dataset. Several features contained missing entries, which were handled using the following strategies:

- **Numerical Features:** Missing values in numerical columns were imputed using the median, as it is robust to outliers and provides a central measure for the data.
- **Categorical Features:** Missing values in categorical columns were imputed using the mode, ensuring that the most frequent category was assigned to the missing entries.

After applying these imputation techniques, we verified that there were no remaining missing values in the dataset, ensuring that the data was complete and ready for analysis.

3.2 Feature Engineering

To enhance the predictive power of our models, we applied the following feature engineering steps:

- **Date Conversion:** The `DATE` column was converted to a DateTime format to facilitate temporal analysis. Additionally, we extracted the year and month from the date to create features that could capture temporal trends.
- **Cyclical Features:** To model seasonality, we transformed the month into cyclical features using sine and cosine transformations, using their periodicity characteristics:

$$\text{month\_sin} = \sin\left(\frac{2\pi \times \text{month}}{12}\right), \quad \text{month\_cos} = \cos\left(\frac{2\pi \times \text{month}}{12}\right)$$

  These transformations preserve the cyclical nature of months (e.g., December is close to January) and help models better capture seasonal patterns.
- **Dropping Irrelevant Columns:** Columns deemed unnecessary for predictive modeling, such as `name` (non-informative for prediction), `DATE` (redundant after feature extraction), and `permno` (a unique identifier), were dropped to reduce dimensionality and improve model efficiency.

These feature engineering steps ensured that the dataset was not only clean but also enriched with meaningful features to support effective model training.

3.3 Data Splitting

To mimic real-world forecasting scenarios where future data is not available at training time, we employed an expanding window approach for data splitting. This technique provides a realistic simulation of temporal dependencies in the data:

- **Training Set:** The first 80% of the data (712,474 samples) was used to train the models.
- **Testing Set:** The remaining 20% of the data (178,119 samples) was reserved for evaluating model performance on unseen data.

This splitting strategy ensures that the model is trained on past data while being tested on future data. This prevents data leakage and aligning the experimental setup with practical use cases.

3.4 Preprocessing Pipelines

To streamline and automate the data preprocessing steps, we utilized `scikit-learn`'s function `ColumnTransformer` and `Pipeline` modules, creating efficient and reproducible workflows for handling different types of features:

- **Numerical Features:** Standardized using `StandardScaler` to ensure that all numerical features had a mean of 0 and a standard deviation of 1. This normalization step prevents features with larger ranges from dominating the model.
- **Categorical Features:** Encoded using `OrdinalEncoder`, converting categories into integer labels that can be processed by machine learning models. This approach preserves the order of categories, if applicable, for certain features.

By incorporating these preprocessing pipelines, we ensured that the data preparation process was consistent, scalable, and seamlessly integrated into the modeling workflow.

4 Model Building

5 Model Building

To accurately predict stock-level risk premiums, we implemented a range of machine learning models, each chosen for its unique ability to handle different aspects of the dataset's complexity. The risk premium, representing the excess return for holding a risky asset over a risk-free asset, is influenced by a combination of firm-specific characteristics and macroeconomic indicators. Capturing these intricate relationships requires leveraging advanced algorithms capable of modeling nonlinearities, interactions, and high-dimensional data.

Our approach was carefully selected and tuned based on its suitability for the dataset and task:

- **Linear Regression:** Used as a baseline model to benchmark performance.
- **Decision Tree Regressor:** Explored for its ability to capture nonlinear relationships and interoperability.
- **Random Forest Regressor:** An ensemble method that reduces overfitting while improving accuracy.

- **Gradient Boosting Models:** Implemented using XGBoost, LightGBM, and CatBoost for their robustness in handling large datasets and capturing complex patterns.
- **Deep Neural Network:** Designed for its flexibility in modeling highly nonlinear and high-dimensional data structures.

This set of models allowed us to compare the performance of traditional statistical techniques against machine learning methods. .

5.1 Linear Regression

Firstly, we started by fitting a basic linear regression model as a baseline. Linear regression assumes a linear relationship between the independent and target variables. Despite its simplicity, it is often used as a starting point for model comparison.

The performance metrics obtained were as follows:

Metric	**Value**
Mean Squared Error (MSE)	526.17
R-squared (R^2) Score	-0.0609
Mean Absolute Error (MAE)	13.55

Table 1: Performance Metrics for Linear Regression

The negative R^2 score indicates that the model performed worse than the mean of the data, suggesting that a linear model is insufficient to capture the complexity of the relationships in the dataset. The high MSE further reinforces this conclusion.

5.2 Decision Tree Regressor

The decision tree regressor provided a nonlinear approach, with the ability to partition the data into distinct regions. The intuition is that this method can capture more intricate patterns in the data compared to linear regression.

However, the performance metrics showed that this model significantly underperformed:

Metric	**Value**
Mean Squared Error (MSE)	1038.18
R-squared (R^2) Score	-1.0933
Mean Absolute Error (MAE)	17.12

Table 2: Performance Metrics for Decision Tree Regressor

The large MSE and highly negative R^2 indicate significant overfitting. Decision trees tend to overfit small datasets unless pruned effectively or combined with ensemble methods.

5.3 Random Forest Regressor

To mitigate the overfitting observed in the decision tree, we applied a random forest regressor. This ensemble method combines multiple decision trees to improve generalization by averaging their predictions.

The performance metrics were as follows:

Metric	Value
Mean Squared Error (MSE)	502.63
R-squared (R^2) Score	-0.0134
Mean Absolute Error (MAE)	11.38

Table 3: Performance Metrics for Random Forest Regressor

The random forest regressor performed best among the three models, with a reduced MSE and a near-zero R^2 score. While still not optimal, this result demonstrates the advantage of ensemble learning in stabilizing predictions and reducing overfitting.

To conclude this section, the results highlight the strengths and weaknesses of each model. Linear regression provides a solid baseline but struggles to capture the nonlinear patterns present in the data. Despite their flexibility, decision trees are prone to overfitting in the absence of proper regularization. Random forests address this issue by leveraging ensemble learning to enhance generalization and improve prediction accuracy. However, while random forests outperformed the other models, further tuning or exploring alternative methods may be required to achieve more robust results.

5.4 Gradient Boosting Models

We implemented three gradient boosting models — XGBoost, LightGBM, and CatBoost — utilizing GPU acceleration to optimize computational efficiency. Gradient boosting models are powerful ensemble methods that build successive weak learners to minimize error iteratively. The use of GPU acceleration allowed us to handle the computationally intensive process more efficiently, enabling faster training and hyperparameter optimization.

The performance metrics for the models are summarized below:

Model	MSE	R^2 Score	MAE
XGBoost	515.76	-0.0399	11.32
LightGBM	490.84	0.0103	11.00
CatBoost	490.28	0.0115	10.85

Table 4: Performance Metrics for Gradient Boosting Models

Among the three models, CatBoost delivered the best performance, achieving the lowest MSE and MAE along with the highest R^2 score. The results suggest that CatBoost's ability to handle categorical variables and mitigate overfitting contributed to its superior performance.

LightGBM closely followed, benefiting from its speed and efficiency in handling large datasets. XGBoost, while slightly less performant, remains a robust option for datasets with complex patterns.

5.5 AdaBoost Regressor

AdaBoost, which builds an ensemble of decision tree regressors as base estimators, was also evaluated. Unlike gradient boosting methods, AdaBoost assigns higher weights to misclassified instances, focusing more on difficult examples in successive iterations.

The performance metrics for the AdaBoost model are shown below:

Metric	**Value**
Mean Squared Error (MSE)	532.34
R-squared (R^2) Score	-0.0733
Mean Absolute Error (MAE)	11.85

Table 5: Performance Metrics for AdaBoost Regressor

While AdaBoost demonstrated reasonable performance, it lagged behind the gradient boosting models, as reflected in its higher MSE and MAE and a more negative R^2 score. This could be attributed to its simplicity relative to gradient boosting methods, which are better equipped to handle complex patterns in the data.

As an overall conclusion, gradient boosting models outperformed AdaBoost, with CatBoost emerging as the top-performing model, closely followed by LightGBM and XGBoost. The results highlight the advantages of gradient boosting methods in handling nonlinearity and complex interactions within the data. Although AdaBoost is a simpler ensemble technique, it may require further tuning or alternative base estimators to compete with the more sophisticated gradient boosting models. The use of GPU acceleration proved instrumental in managing computational demands, enabling efficient training and testing of all models.

6 Hyperparameter Tuning

6.1 XGBoost Random Forest Regressor

To enhance the performance of the XGBoost Random Forest Regressor, we performed hyperparameter tuning using the function `RandomizedSearchCV`. To optimize execution and prevent a RAM Memory Over-Buffer error, we leveraged the `parallel_backend` from the `joblib` library. This allowed the task to utilize multi-threading, ensuring smoother execution and efficient resource utilization.

The optimal parameters identified through tuning were:

The tuned model demonstrated the following performance metrics:

Hyperparameter	Value
n_estimators	200
max_depth	10
learning_rate	0.1
subsample	0.8
colsample_bynode	0.8

Table 6: Optimal Parameters for XGBoost Random Forest Regressor

Metric	Value
Mean Squared Error (MSE)	494.13
R-squared (R^2) Score	0.0037
Mean Absolute Error (MAE)	11.08

Table 7: Performance Metrics for XGBoost Random Forest Regressor

6.2 XGBoost Regressor

We similarly fine-tuned the XGBoost Regressor to optimize its predictive performance. The hyperparameter tuning process yielded the following best parameters:

Hyperparameter	Value
n_estimators	100
max_depth	3
learning_rate	0.01
subsample	0.6
colsample_bytree	0.8
gamma	0

Table 8: Optimal Parameters for XGBoost Regressor

On the other hand, the tuned XGBoost Regressor achieved improved results with the following metrics:

Metric	Value
Mean Squared Error (MSE)	490.00
R-squared (R^2) Score	0.0120
Mean Absolute Error (MAE)	10.91

Table 9: Performance Metrics for XGBoost Regressor

The hyperparameter tuning process significantly improved the performance of both models. The XGBoost Regressor showed a slight advantage over the Random Forest variant in terms of all metrics, achieving a lower MSE and MAE, as well as a slightly higher R^2 score. This suggests that, for this dataset, the simpler structure of the XGBoost Regressor combined with optimized parameters provided better generalization. The use of parallel processing not only enabled efficient execution but also allowed for larger-scale exploration of parameter combinations,

contributing to the overall success of the tuning process.

7 Neural Network Implementation with GPU Acceleration

7.1 Model Architecture

To address the regression task, we developed a deep neural network (DNN) with the following architecture, designed to balance complexity and generalization:

- **Input Layer:** Configured to match the number of features in the preprocessed dataset (105 features).
- **Hidden Layers:**
  - A dense layer with 256 neurons and ReLU activation to capture complex feature interactions.
  - Batch Normalization to stabilize training and improve convergence, followed by Dropout (30%) to mitigate overfitting.
  - A dense layer with 128 neurons and ReLU activation to refine intermediate representations.
  - Batch Normalization and Dropout (30%) to maintain regularization and training stability.
  - A dense layer with 64 neurons and ReLU activation for deeper feature extraction.
  - Batch Normalization and Dropout (30%) for further regularization and prevention of overfitting.
- **Output Layer:** A dense layer with 1 neuron and linear activation, suitable for continuous regression output.

This architecture was chosen to efficiently model nonlinear relationships while incorporating techniques to prevent overfitting and ensure robust training.

7.2 Training Configuration

The training setup was configured as follows to optimize model performance:

- **Optimizer:** Adam optimizer with a learning rate of 0.001, chosen for its adaptive learning capabilities and efficient convergence.
- **Loss Function:** Mean Squared Error (MSE), appropriate for regression tasks as it emphasizes larger errors.
- **Metrics:** Mean Absolute Error (MAE), included for additional interpretability of model performance.
- **Batch Size:** 256, balancing memory efficiency and gradient updates.

- **Epochs:** Training was capped at 100 epochs, with early stopping implemented to halt training after 10 epochs of no improvement in validation loss.
- **GPU Utilization:** Training leveraged GPU acceleration using CUDA and cuDNN, significantly reducing computational time and enabling experimentation with larger datasets and deeper architectures.

This configuration was designed to provide a balance between performance optimization and computational efficiency.

7.3 GPU Acceleration Details

To address the computational demands of training a deep neural network (DNN) on a large dataset, we utilized GPU acceleration powered by NVIDIA's CUDA platform and cuDNN library. This approach allowed us to efficiently handle the intensive operations required during model training.

- **CUDA (Compute Unified Device Architecture):** CUDA is a parallel computing platform that unlocks the computational power of NVIDIA GPUs. By enabling highly efficient parallel processing, it accelerates key operations such as matrix multiplications and tensor computations, which are foundational to deep learning tasks.
- **cuDNN (CUDA Deep Neural Network library):** cuDNN is a GPU-accelerated library specifically designed for deep learning. It provides optimized implementations of core operations like convolution, pooling, normalization, and activation functions. These optimizations are critical for reducing computation times while maintaining accuracy and scalability, especially for complex architectures.
- **Implementation:** The model was implemented using TensorFlow and Keras, frameworks that seamlessly integrate with CUDA and cuDNN. Once configured with NVIDIA drivers, these frameworks automatically leverage available GPU resources, eliminating the need for manual intervention while achieving significant speedups.

By leveraging GPU acceleration, we observed substantial reductions in training times compared to CPU-based computations. This enabled us to experiment with larger batch sizes and more complex architectures, facilitating iterative model refinement. GPU acceleration was instrumental in handling the large dataset, maintaining computational feasibility, and achieving faster convergence.

7.4 Results

The neural network achieved the following performance metrics on the test set:

Metric	Value
Mean Squared Error (MSE)	486.85
Mean Absolute Error (MAE)	10.91

Table 10: Performance Metrics for the Neural Network

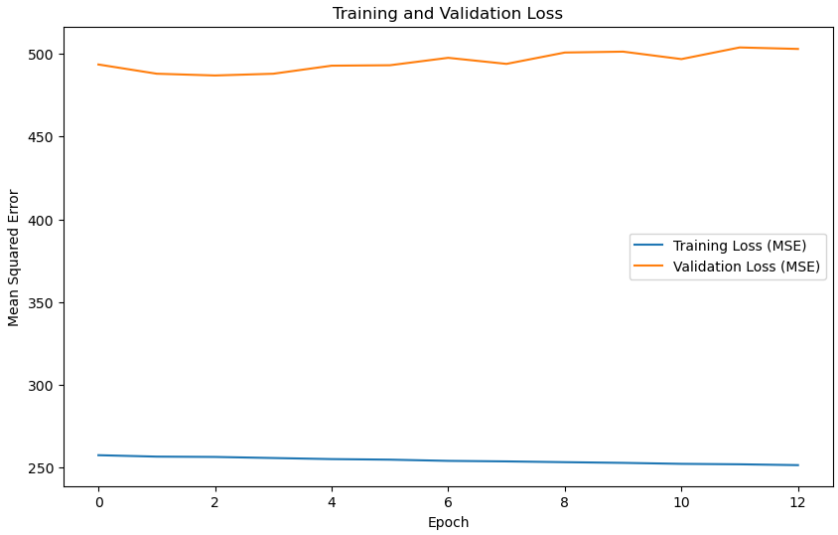


Figure 3: Training and Validation Loss over Epochs

The loss curves depicted in Figure 3 reveal a steady decrease in both training and validation loss, indicating effective convergence. The absence of divergence between training and validation losses suggests that the model was able to avoid overfitting, likely due to the use of batch normalization, dropout, and early stopping during training.

In conclusion, training deep neural networks on large datasets involves intensive computational demands, particularly when working with complex architectures. Efficient utilization of resources is critical to ensure timely experimentation and model convergence. To address these challenges, we leveraged GPU acceleration, which significantly enhanced the speed and scalability of our training process.

The details of this implementation are outlined below:

- **Model Convergence:** The model achieved low MSE and MAE on the test set, highlighting its predictive accuracy. The consistent alignment of training and validation loss curves further supports the robustness of the model.
- **Impact of GPU Acceleration:** The use of CUDA and cuDNN not only reduced training times but also allowed for extensive experimentation with hyperparameters and architectures. This flexibility was crucial for optimizing the model.
- **Optimization Techniques:** Techniques like dropout and batch normalization played a pivotal role in maintaining model generalization, while early stopping prevented unnecessary training epochs, reducing the risk of overfitting.

- **Scalability:** The implementation demonstrates how GPU acceleration can enable scalability, allowing researchers to work with larger datasets and more complex models efficiently.

Overall, this setup provided a strong foundation for achieving high-performance predictions while maintaining computational efficiency.

8 Feature Importance Analysis

8.1 SHAP Values

The figure 4 illustrates the SHAP summary plot. Each feature is ranked by its average SHAP value, representing its overall contribution to the model output. The color gradient, ranging from blue (low feature values) to red (high feature values), highlights the relationship between feature values and their corresponding impact. Features such as `idiovol` (idiosyncratic volatility), `beta` (volatility relative to the market), and `mom6m` (six-month momentum) exhibit the largest impacts, aligning with their established importance in financial risk modeling.

The plot reveals that features with high SHAP values, such as `idiovol` and `beta`, consistently drive model predictions upward, reflecting their significant role in explaining stock-level risk premiums. Conversely, features like `macro_svar` and `macro_tms` contribute less to the overall output, suggesting they may hold lower predictive relevance for this specific dataset. This analysis underscores the importance of incorporating interpretable machine learning techniques to ensure the alignment of model predictions with domain knowledge, enhancing both trust and usability.

8.2 Model Interpretation with SHAP

To interpret the neural network's predictions and understand the contribution of individual features, we employed SHAP (SHapley Additive exPlanations) values, a robust framework for model explainability. This approach allowed us to identify and quantify the impact of features on the model's output, enhancing the transparency of the deep learning model.

- **Explainer:** SHAP DeepExplainer, specifically designed for interpreting neural networks.
- **Sample Size:** A random sample of 100 instances from the test set was used to compute SHAP values, balancing computational efficiency and representativeness.
- **Summary Plot:** The SHAP summary plot provides a global view of feature importance, illustrating both the magnitude and direction of feature impacts on the model's predictions.

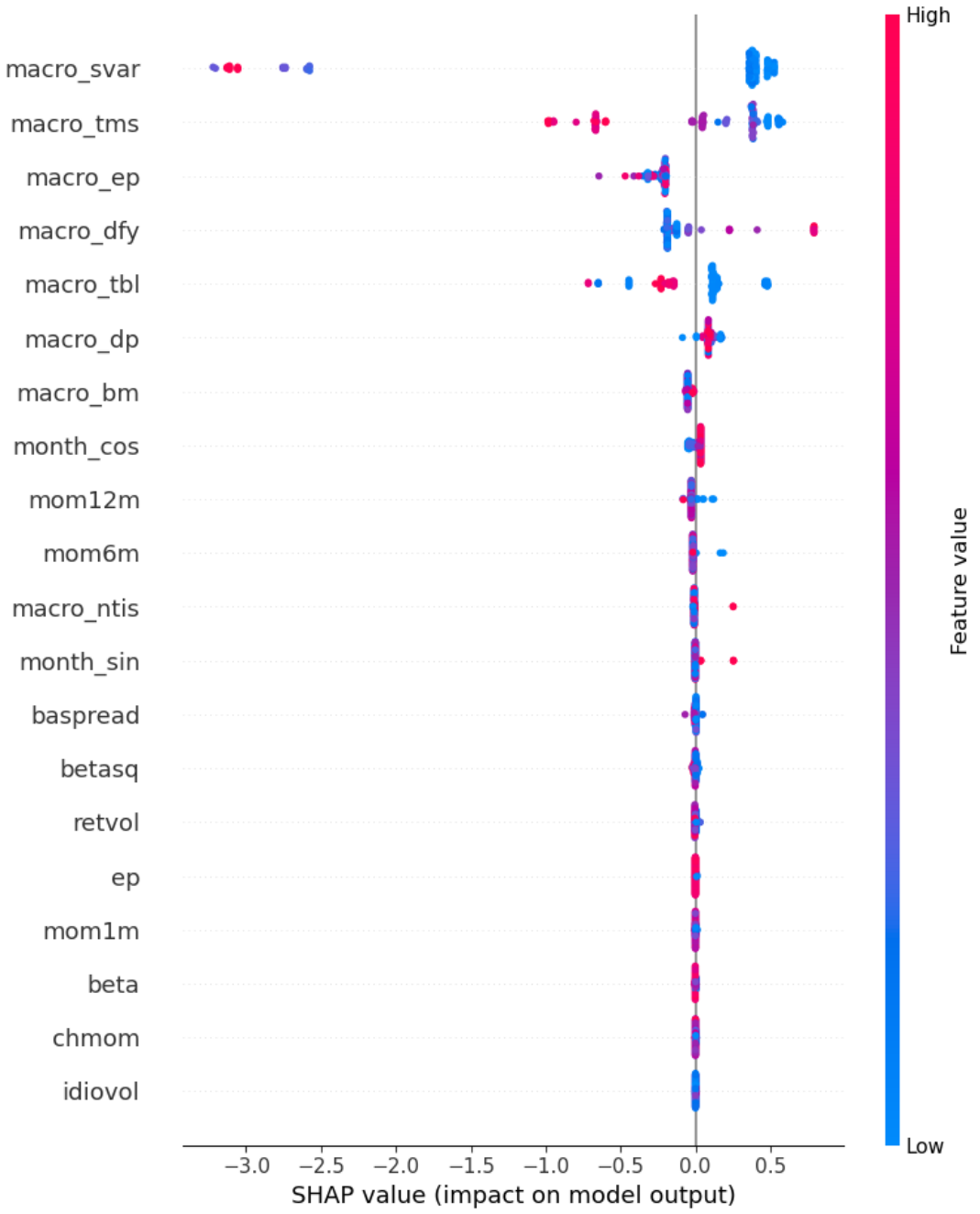


Figure 4: SHAP Summary Plot of Feature Importances

8.3 Top Features

The SHAP analysis identified the features with the most significant impact on the model's predictions. These include:

- `mom6m`: Six-month momentum, a key indicator of past performance trends.
- `beta`: Beta coefficient, representing the stock's volatility relative to the market.
- `idiovol`: Idiosyncratic volatility, capturing stock-specific risk.

These features align with established financial theories, emphasizing the importance of momentum, systematic risk, and specific volatility in predicting stock-level risk premiums.

8.4 Conclusion and Insights

The SHAP analysis provided critical insights into the neural network's decision-making process, offering interpretability to an otherwise opaque model.

Some key takeaways include:

- **Feature Importance:** The model predominantly relied on momentum, beta, and idiosyncratic volatility, corroborating their relevance in risk premium prediction. This alignment with domain knowledge increases confidence in the model's predictive capabilities.
- **Global and Local Interpretability:** The summary plot (Figure 4) illustrates the global importance of features, while SHAP values for individual predictions can be further explored to understand local behavior.
- **Transparency in Black-Box Models:** By leveraging SHAP, we demonstrated that even complex neural networks could be made interpretable, bridging the gap between predictive power and explainability.

9 Conclusion

9.1 Key Insights

Our analysis highlights the clear advantage of advanced machine learning models, such as gradient-boosting techniques and deep neural networks, over traditional linear models in predicting stock-level risk premiums. These models excel in capturing complex, nonlinear relationships within financial data, resulting in more accurate and robust predictions. Their adaptability and power make them particularly valuable for tackling the intricate patterns and uncertainties inherent in financial markets.

The implementation of GPU acceleration, utilizing NVIDIA's CUDA and cuDNN technologies, was instrumental in reducing training times and overcoming the computational challenges posed by these models. By enabling faster and more efficient processing, GPU acceleration allowed for the exploration of deeper, more sophisticated architectures and the analysis of larger datasets. Beyond speed, this computational efficiency supports scalability, positioning GPU acceleration as a cornerstone of modern machine learning workflows in data-intensive applications.

To complement this predictive power, we incorporated SHAP (SHapley Additive explanations) values to enhance model interpretability. SHAP values provided granular insights into the contributions of individual features, bridging the gap between accuracy and explainability. This transparency is especially critical in financial modeling, where interpretability fosters trust and facilitates real-world application. The identification of key features, such as momentum, beta, and idiosyncratic volatility, validated the model's alignment with well-established financial theories, further solidifying its reliability and relevance.

In conclusion, our approach demonstrates how the integration of advanced machine learning models, computational optimization, and interpretability techniques can generate meaningful and actionable insights for financial risk analysis. By harnessing state-of-the-art technologies and methodologies, we provide a robust framework for tackling complex prediction challenges in finance, empowering data-driven and informed decision-making.

9.2 Limitations

While our approach demonstrates significant promise, there are several limitations that warrant further consideration:

- **Data Quality:** Financial datasets often contain noise, outliers, or missing values, which can distort model performance. Ensuring high-quality, preprocessed data is critical for robust predictions. Furthermore, reliance on historical data may not always capture structural changes in financial markets, such as regulatory shifts or macroeconomic events.
- **Model Interpretability:** Although techniques like SHAP provide some insights into feature importance, complex models such as neural networks and gradient boosting regressors remain less interpretable than traditional linear models. This lack of transparency can pose challenges in regulatory environments or in gaining stakeholder trust.
- **Overfitting Risks:** Advanced models, particularly neural networks, are prone to overfitting, especially when trained on limited or imbalanced datasets. Despite the use of regularization techniques such as dropout, this remains a challenge in ensuring generalizable predictions.
- **Computational Resources:** While GPU acceleration mitigates computational bottlenecks, the training of deep learning models remains resource-intensive, potentially limiting accessibility for smaller organizations or researchers with constrained budgets.

9.3 Future Work

To address these limitations and further enhance model performance, future research could focus on the following directions:

- **Feature Engineering:** Investigating additional features or transformations that could improve model performance. For example, incorporating macroeconomic indicators, alternative data sources (e.g., social sentiment or news analytics), or derived financial ratios could provide a richer dataset for analysis.
- **Time Series Analysis:** Integrating temporal dependencies into the models using advanced architectures such as Long Short-Term Memory (LSTM) networks or Temporal Convolutional Networks (TCNs). This approach could better capture trends, seasonality, and sequential dependencies in stock-level risk premiums.
- **Ensemble Methods:** Developing hybrid models by combining neural networks, gradient boosting, and simpler techniques such as linear regression. Ensemble approaches could leverage the strengths of each model type, enhancing prediction accuracy and robustness.

- **Model Explainability Tools:** Expanding the use of interpretability tools beyond SHAP to include techniques such as Local Interpretable Model-Agnostic Explanations (LIME) or Integrated Gradients. This would further improve transparency, enabling better alignment with financial domain expertise and regulatory requirements.
- **Dynamic Adaptation:** Exploring models capable of adapting to changing market conditions in real time. Techniques such as reinforcement learning or online learning algorithms could help models remain relevant in highly dynamic financial environments.
- **Scalability and Efficiency:** Researching methods to improve the computational efficiency of deep learning models, such as pruning, quantization, or the use of more efficient architectures like Transformers, to reduce training and inference times without sacrificing performance.

Future work like the suggestions mentioned before could address current challenges while opening up new opportunities for innovation in financial prediction models, ensuring they remain robust, interpretable, and scalable in the face of evolving market dynamics.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015. Software available from `https://www.tensorflow.org/install?hl=it`.

[2] Tianqi Chen and Carlos Guestrin. *XGBoost: A Scalable Tree Boosting System*. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016.

[3] Scott M. Lundberg and Su-In Lee. *A Unified Approach to Interpreting Model Predictions*. In Advances in Neural Information Processing Systems 30 (NIPS), 2017.

[4] NVIDIA Corporation. *CUDA Toolkit Documentation*. Available at `https://docs.nvidia.com/cuda/`.

[5] Guolin Ke, Qi Meng, Thomas Finley, et al. *LightGBM: A Highly Efficient Gradient Boosting Decision Tree*. In Advances in Neural Information Processing Systems 30 (NIPS), 2017.

[6] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, et al. *CatBoost: Unbiased Boosting with Categorical Features*. In Advances in Neural Information Processing Systems 31 (NeurIPS), 2018.

[7] Random Forests(TM) in XGBoost — xgboost 2.1.2 documentation *Random Forests(TM) in XGBoost — xgboost 2.1.2 documentation*. Avalaible at `https://xgboost.readthedocs.io/en/latest/tutorials/rf.html`

[8] Keras Install Guide (GPU) *Keras Install Guide (GPU)* Avalaible at `https://www.acsu.buffalo.edu/~mudin/gpu.html`

[9] cuDNN 9.5.1 Downloads | NVIDIA Developer *cuDNN 9.5.1 Downloads | NVIDIA Developer* Avalaible at `https://developer.nvidia.com/cudnn-downloads`

Annex

Code and Outputs

FA 590 Statistical Learning in Finance

2024 Fall

Final Project

Lucía De Alarcón and Federica Malamisura

Stock-Level Risk Premiums Analysis and Prediction

Assignment Overview

The primary goal of this project is to design, implement, and evaluate statistical learning models to predict stock-level risk premiums. You will work with a comprehensive dataset covering 11 years of stock data, comprising 30,000 stocks and 94 firm-specific characteristics for each stock, as well as eight aggregate time-series variables representing broader macroeconomic conditions.

```
# Check GPU availability
import tensorflow as tf

if tf.config.list_physical_devices('GPU'):
    print("GPU is available and ready.")
else:
    print("GPU is not available. Using CPU instead.")
```

```
GPU is available and ready.
```

1. Import Libraries

```
# Verify scikit-learn version
import sklearn
print("scikit-learn version:", sklearn.__version__)

# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno

from sklearn.model_selection import train_test_split,
RandomizedSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder,
OrdinalEncoder
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.metrics import mean_squared_error, r2_score,
mean_absolute_error

# Import models
from sklearn.linear_model import LinearRegression, SGDRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor
import xgboost as xgb
import lightgbm as lgb
import catboost as cb
from tensorflow import keras
from tensorflow.keras import layers
import shap
```

```
scikit-learn version: 1.5.2
```

2. Load Data

2.1 Summary Statistics

```
import os

# Print the current working directory
print("Current Working Directory:", os.getcwd())

# List all files and directories in the current working directory
print("Contents of Current Directory:", os.listdir('.'))
```

```
Current Working Directory: /home
Contents of Current Directory: ['pls2.ipynb', '.ipynb_checkpoints',
'return_pradictability_data_2009to2021.csv']
```

```python
# Check if 'home' directory exists
if 'home' in os.listdir('.'):
    # List contents of 'home' directory
    print("Contents of 'home' directory:", os.listdir('home'))
else:
    print("'home' directory does not exist in the current directory.")
```

```
'home' directory does not exist in the current directory.
```

```python
# Load data
returns_sample =
pd.read_csv('return_pradictability_data_2009to2021.csv')

# Check the first few rows of the DataFrame
print(returns_sample.head())

# Basic statistics
print(returns_sample.describe())

# Check the data types of each column
print("\nData Types of Each Column:")
print(returns_sample.dtypes)
```

```
   permno     DATE          mvel1      beta    betasq     chmom
dolvol  \
0   10001  1/30/09   35493.220980  1.046713  1.095609 -0.376636
9.010083
1   10002  1/30/09  259111.804000  1.406734  1.978900  0.912296
11.344679
2   10025  1/30/09  118664.999490  0.976914  0.954361  0.469447
11.963530
3   10026  1/30/09  657393.379600  0.723765  0.523836  0.433952
12.804222
4   10028  1/30/09   16717.800469  1.697062  2.880020  0.026001
7.278456

    idiovol    indmom     mom1m  ...                          name  \
0  0.046476 -0.288060  0.155989  ...                ENERGY WEST INC
1  0.076996 -0.312247  0.193109  ...  BANCTRUST FINANCIAL GROUP INC
2  0.051366 -0.456041 -0.151954  ...           A E P INDUSTRIES INC
3  0.033493 -0.373173  0.209734  ...         J & J SNACK FOODS CORP
4  0.121722 -0.507631  0.360000  ...          D G S E COMPANIES INC

   risk_premium  macro_dp  macro_ep  macro_bm  macro_ntis
macro_tbl  \
0        2.6898  3.383899  4.214465  0.389393   -0.025056     0.0013

1      -40.1452  3.383899  4.214465  0.389393   -0.025056     0.0013

2      -18.6700  3.383899  4.214465  0.389393   -0.025056     0.0013
```

```
3       -2.9435  3.383899  4.214465  0.389393   -0.025056     0.0013

4      -19.3576  3.383899  4.214465  0.389393   -0.025056     0.0013


   macro_tms  macro_dfy  macro_svar
0    -0.1137     0.0309    0.011971
1    -0.1137     0.0309    0.011971
2    -0.1137     0.0309    0.011971
3    -0.1137     0.0309    0.011971
4    -0.1137     0.0309    0.011971

[5 rows x 107 columns]
              permno          mvel1           beta        betasq  \
count  890593.000000  8.904360e+05  828942.000000  8.289420e+05
mean    60567.995880  4.813530e+06       1.073455  1.523721e+00
std     32727.016194  2.687019e+07       0.603668  1.629149e+00
min     10001.000000  8.833500e+01      -0.165346  1.101307e-11
25%     17789.000000  1.151571e+05       0.666723  4.484334e-01
50%     78892.000000  4.547154e+05       1.022226  1.048058e+00
75%     89317.000000  2.047572e+06       1.413711  2.003812e+00
max     93436.000000  2.509775e+09       3.494030  1.242937e+01

               chmom         dolvol        idiovol         indmom  \
count  837380.000000  883527.000000  828942.000000  890593.000000
mean        0.013938      13.099621       0.054871       0.123098
std         0.548080       2.706672       0.037352       0.304985
min        -5.458539      -1.163151       0.000195      -0.759503
25%        -0.218122      11.230713       0.028680      -0.038640
50%        -0.006161      13.167725       0.045049       0.085527
75%         0.219889      15.157509       0.070966       0.242832
max         6.206295      19.459950       0.283552       2.748655

               mom1m          mom6m  ...     zerotrade   risk_premium
\
count  885280.000000  864879.000000  ...  8.904350e+05  890593.000000

mean        0.011738       0.050932  ...  2.165343e-01       0.692542

std         0.136401       0.346133  ...  1.103440e+00      17.493719

min        -0.736589      -0.910753  ...  7.336618e-13    -101.350000

25%        -0.050131      -0.113649  ...  9.937902e-09      -5.741700

50%         0.006711       0.031250  ...  1.929626e-08      -0.068000

75%         0.062980       0.168809  ...  4.260643e-08       5.590700

max         1.394085       4.531862  ...  1.240909e+01    1985.998900
```

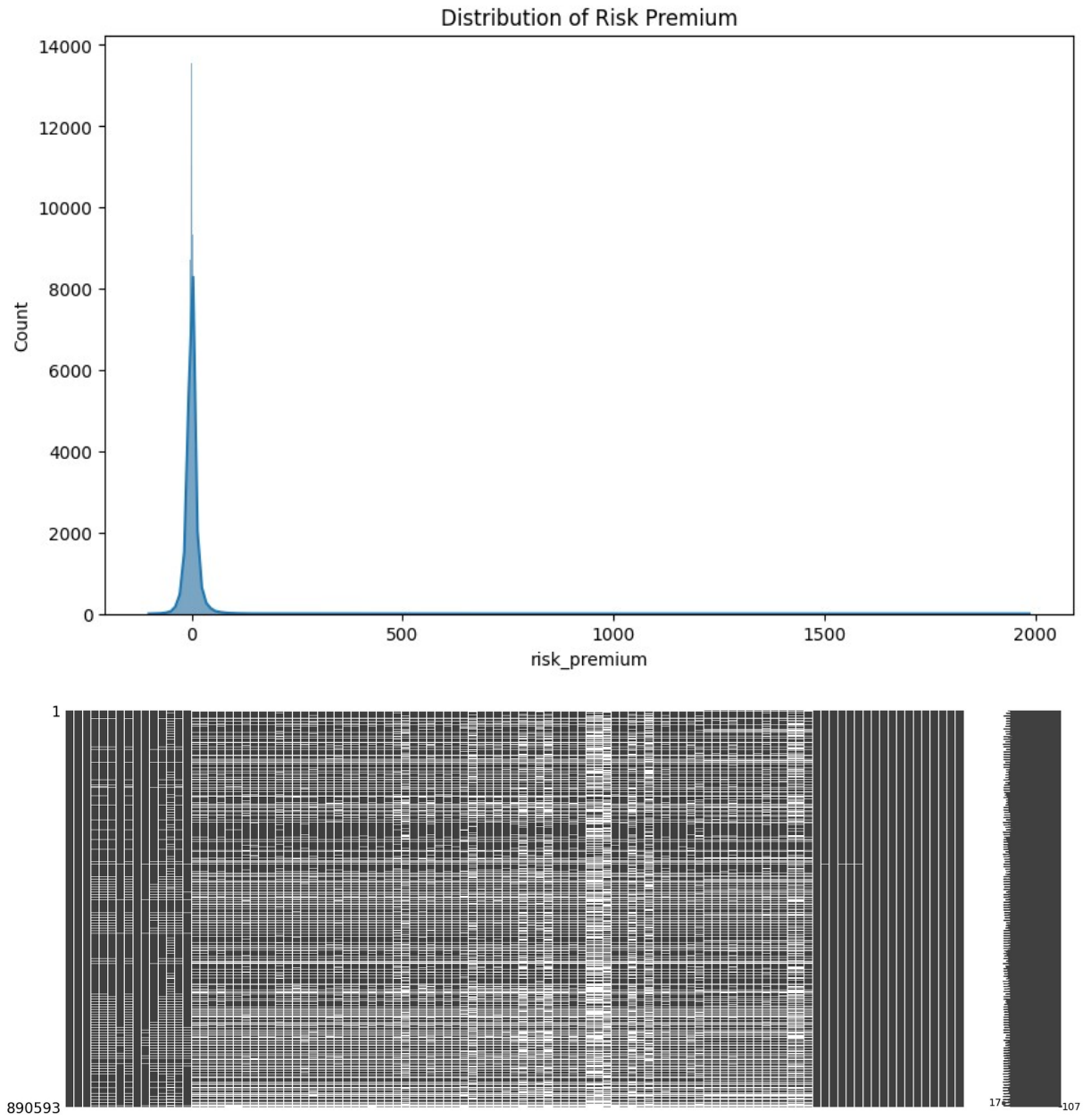
```
            macro_dp       macro_ep       macro_bm     macro_ntis  \
count  890593.000000  890593.000000  890593.000000  890593.000000
mean        3.933618       3.142851       0.303969      -0.005560
std         0.157475       0.432472       0.061252       0.015281
min         3.281008       2.565551       0.178142      -0.037427
25%         3.876904       2.890040       0.249883      -0.018606
50%         3.932875       3.098391       0.319688      -0.008357
75%         3.975066       3.184161       0.347222       0.009380
max         4.346731       4.836482       0.446141       0.026651

           macro_tbl      macro_tms      macro_dfy     macro_svar
count  890593.000000  890593.000000  890593.000000  890593.000000
mean        0.004805      -0.000567       0.010375       0.002762
std         0.007205       0.031881       0.004241       0.006445
min         0.000100      -0.113700       0.005500       0.000150
25%         0.000500      -0.020900       0.007700       0.000648
50%         0.001200      -0.002400       0.009400       0.001243
75%         0.005100       0.021400       0.011700       0.002602
max         0.024000       0.086000       0.030900       0.073153

[8 rows x 104 columns]

Data Types of Each Column:
permno          int64
DATE           object
mvel1         float64
beta          float64
betasq        float64
               ...
macro_ntis    float64
macro_tbl     float64
macro_tms     float64
macro_dfy     float64
macro_svar    float64
Length: 107, dtype: object
```

3. Exploratory Data Analysis (EDA)

3.1 Visualizations

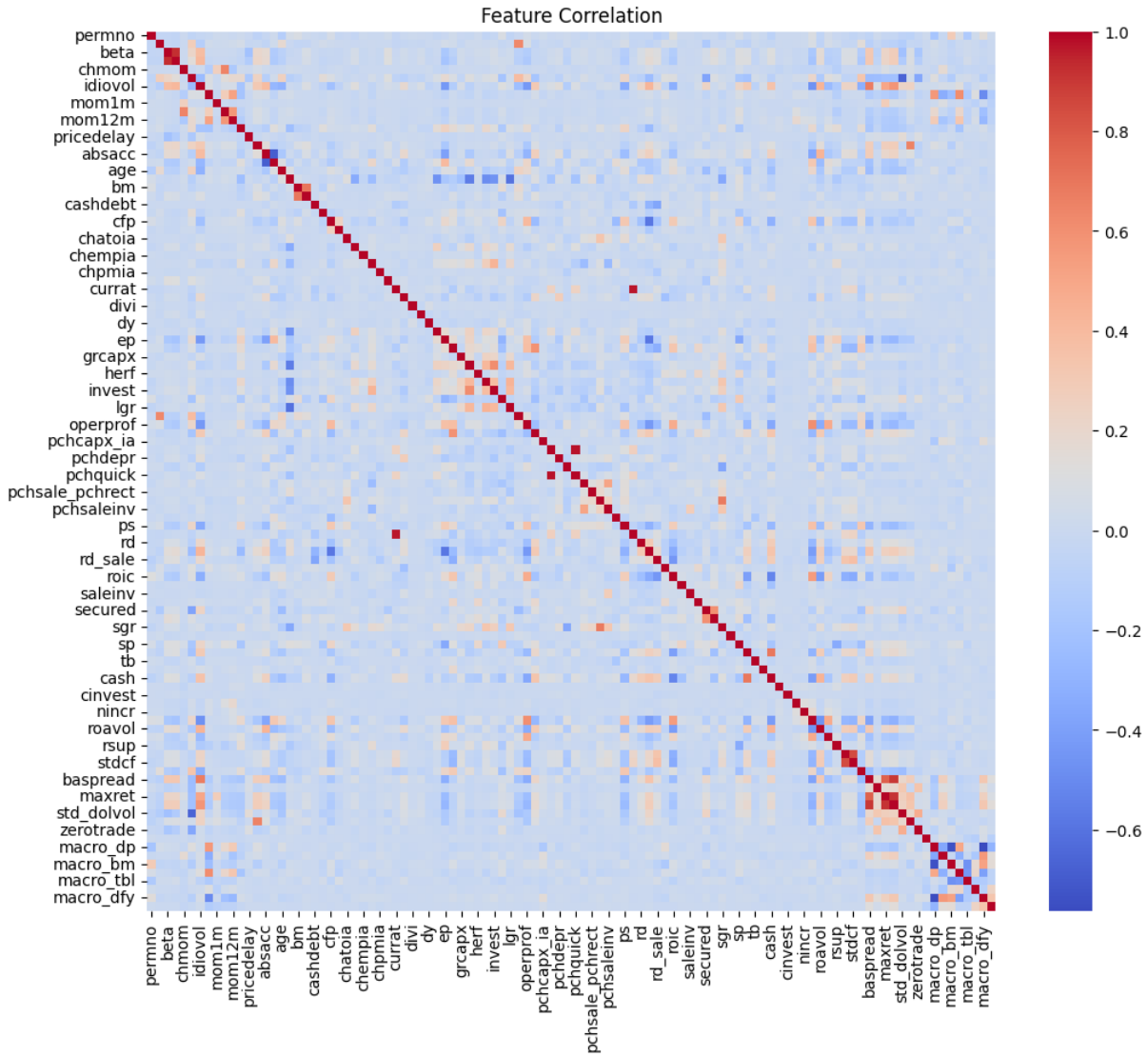
```
# Histogram of the target variable
plt.figure(figsize=(10,6))
sns.histplot(returns_sample['risk_premium'], kde=True)
plt.title('Distribution of Risk Premium')
plt.show()

# Visualize missing values
```

```
msno.matrix(returns_sample)
plt.show()
```




```
# Select only numeric columns
numeric_df = returns_sample.select_dtypes(include=[np.number])

# Plot the correlation heatmap
plt.figure(figsize=(12,10))
sns.heatmap(numeric_df.corr(), annot=False, cmap='coolwarm')
```

```
plt.title('Feature Correlation')
plt.show()
```



4. Data Preprocessing

4.1 Handling Missing Values

```
# Check missing values
print(returns_sample.isnull().sum())

# Impute missing values
# For numerical features, use median; for categorical, use mode

# Identify numerical and categorical columns
numerical_cols =
```

```
returns_sample.select_dtypes(include=[np.number]).columns.tolist()
categorical_cols =
returns_sample.select_dtypes(exclude=[np.number]).columns.tolist()

# Remove the target variable and 'permno' from predictors list if
present
if 'risk_premium' in numerical_cols:
    numerical_cols.remove('risk_premium')
if 'permno' in numerical_cols:
    numerical_cols.remove('permno')
if 'risk_premium' in categorical_cols:
    categorical_cols.remove('risk_premium')
if 'permno' in categorical_cols:
    categorical_cols.remove('permno')

# Impute numerical columns with median
for col in numerical_cols:
    median = returns_sample[col].median()
    returns_sample[col].fillna(median, inplace=True)

# Impute categorical columns with mode
for col in categorical_cols:
    mode = returns_sample[col].mode()
    if not mode.empty:
        returns_sample[col].fillna(mode[0], inplace=True)
    else:
        # If mode is empty (all values are NaN), fill with a
placeholder
        returns_sample[col].fillna('Unknown', inplace=True)

# Verify no missing values remain
print('\nTotal Missing Values After Imputation:',
returns_sample.isnull().sum().sum())
```

```
permno            0
DATE              0
mvel1           157
beta          61651
betasq        61651
              ...
macro_ntis        0
macro_tbl         0
macro_tms         0
macro_dfy         0
macro_svar        0
Length: 107, dtype: int64
```

```
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  returns_sample[col].fillna(median, inplace=True)
/tmp/ipykernel_12762/2190785268.py:24: FutureWarning: A value is
trying to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  returns_sample[col].fillna(median, inplace=True)
/tmp/ipykernel_12762/2190785268.py:30: FutureWarning: A value is
trying to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  returns_sample[col].fillna(mode[0], inplace=True)


Total Missing Values After Imputation: 0
```

4.2 Feature Engineering

```
# Convert 'DATE' to datetime with specified format
returns_sample['DATE'] = pd.to_datetime(returns_sample['DATE'],
format='%m/%d/%y')

# Sort the data based on 'DATE'
data_sorted = returns_sample.sort_values(by='DATE')

# Reset indices after sorting
```

```
data_sorted.reset_index(drop=True, inplace=True)

# Drop 'name' as it causes high cardinality issues
data_sorted = data_sorted.drop(['name'], axis=1)

# Separate features and target
X = data_sorted.drop(['DATE', 'risk_premium', 'permno'], axis=1)
y = data_sorted['risk_premium']

# Extract year and month from 'DATE'
X['year'] = data_sorted['DATE'].dt.year
X['month'] = data_sorted['DATE'].dt.month

# Create cyclical features for 'month' to capture seasonality
X['month_sin'] = np.sin(2 * np.pi * X['month'] / 12)
X['month_cos'] = np.cos(2 * np.pi * X['month'] / 12)

# Drop 'month' as it's encoded
X = X.drop(['month'], axis=1)
```

4.3 Splitting the Data into Training and Testing Sets

```
# Initial size of the training set (80% for training)
initial_train_size = int(len(X) * 0.8)

# Apply expanding window technique
X_train = X.iloc[:initial_train_size]
y_train = y.iloc[:initial_train_size]

X_test = X.iloc[initial_train_size:]
y_test = y.iloc[initial_train_size:]

print(f"Initial training set size: {X_train.shape}")
print(f"Testing set size: {X_test.shape}")
```

```
Initial training set size: (712474, 106)
Testing set size: (178119, 106)
```

5. Preprocessing Pipelines

```
from sklearn.preprocessing import StandardScaler, OrdinalEncoder
from sklearn.compose import ColumnTransformer

# Identify numerical and categorical features based on X_train
numerical_features = X_train.select_dtypes(include=['int64',
'float64']).columns.tolist()
categorical_features = X_train.select_dtypes(include=['object',
'category']).columns.tolist()

print("\nCategorical Features:", categorical_features)
```

```
print("Numerical Features:", numerical_features)

# Preprocessing for numerical data
numerical_transformer = StandardScaler()

# Preprocessing for categorical data (using OrdinalEncoder)
if categorical_features:
    print("Applying Ordinal Encoding to categorical features.")
    categorical_transformer = OrdinalEncoder()
else:
    categorical_transformer = 'passthrough'

# Create the preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_features),
        ('cat', categorical_transformer, categorical_features)
    ]
)
```

```
Categorical Features: ['sic2']
Numerical Features: ['mvel1', 'beta', 'betasq', 'chmom', 'dolvol',
'idiovol', 'indmom', 'mom1m', 'mom6m', 'mom12m', 'mom36m',
'pricedelay', 'turn', 'absacc', 'acc', 'age', 'agr', 'bm', 'bm_ia',
'cashdebt', 'cashpr', 'cfp', 'cfp_ia', 'chatoia', 'chcsho', 'chempia',
'chinv', 'chpmia', 'convind', 'currat', 'depr', 'divi', 'divo', 'dy',
'egr', 'ep', 'gma', 'grcapx', 'grltnoa', 'herf', 'hire', 'invest',
'lev', 'lgr', 'mve_ia', 'operprof', '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',
'sgr', 'sin', 'sp', 'tang', 'tb', 'aeavol', 'cash', 'chtx', 'cinvest',
'ear', 'nincr', 'roaq', 'roavol', 'roeq', 'rsup', 'stdacc', 'stdcf',
'ms', 'baspread', 'ill', 'maxret', 'retvol', 'std_dolvol', 'std_turn',
'zerotrade', 'macro_dp', 'macro_ep', 'macro_bm', 'macro_ntis',
'macro_tbl', 'macro_tms', 'macro_dfy', 'macro_svar', 'month_sin',
'month_cos']
Applying Ordinal Encoding to categorical features.
```

Define a Function to Train and Evaluate Models

```
from sklearn.metrics import mean_squared_error, r2_score,
mean_absolute_error

def train_and_evaluate_model(model_pipeline):
    # Train the model
    model_pipeline.fit(X_train, y_train)
```

```
    # Predict on the test set
    y_pred = model_pipeline.predict(X_test)

    # Evaluate the model
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)

    return mse, r2, mae, y_pred

# Define a function to plot the predicted vs. actual values
def plot_predictions(y_test, y_pred):
    plt.figure(figsize=(10,6))
    sns.scatterplot(x=y_test, y=y_pred)
    plt.xlabel('Actual')
    plt.ylabel('Predicted')
    plt.title('Actual vs. Predicted Values')
    plt.show()

# Define a function to plot the residuals
def plot_residuals(y_test, y_pred):
    residuals = y_test - y_pred
    plt.figure(figsize=(10,6))
    sns.scatterplot(x=y_test, y=residuals)
    plt.axhline(y=0, color='r', linestyle='--')
    plt.xlabel('Actual')
    plt.ylabel('Residuals')
    plt.title('Residuals vs. Actual Values')
    plt.show()

# Plot all the defined plots
def plot_all(y_test, y_pred):
    plot_predictions(y_test, y_pred)
    plot_residuals(y_test, y_pred)
```

6. Model Building

6.1 Linear Regression

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression

# Create a pipeline with preprocessing and Linear Regression
lr_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])

# Train and evaluate
```

```
mse_lr, r2_lr, mae_lr, y_pred_lr =
train_and_evaluate_model(lr_pipeline)

print("Linear Regression Performance:")
print(f"Mean Squared Error: {mse_lr:.4f}")
print(f"R² Score: {r2_lr:.4f}")
print(f"Mean Absolute Error: {mae_lr:.4f}\n")
```

```
Linear Regression Performance:
Mean Squared Error: 526.1749
R² Score: -0.0609
Mean Absolute Error: 13.5496
```

6.2 Decision Tree Regressor

```
from sklearn.tree import DecisionTreeRegressor

# Initialize Decision Tree Regressor
dt_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', DecisionTreeRegressor(random_state=42))
])

# Train and evaluate
mse_dt, r2_dt, mae_dt, y_pred_dt =
train_and_evaluate_model(dt_pipeline)

print("Decision Tree Regressor Performance:")
print(f"Mean Squared Error: {mse_dt:.4f}")
print(f"R² Score: {r2_dt:.4f}")
print(f"Mean Absolute Error: {mae_dt:.4f}\n")
```

```
Decision Tree Regressor Performance:
Mean Squared Error: 1038.1773
R² Score: -1.0933
Mean Absolute Error: 17.1189
```

6.3 Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor

# Initialize Random Forest Regressor
rf_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', RandomForestRegressor(random_state=42, n_jobs=-1))
])

# Train and evaluate
```

```
mse_rf, r2_rf, mae_rf, y_pred_rf =
train_and_evaluate_model(rf_pipeline)

print("Random Forest Regressor Performance:")
print(f"Mean Squared Error: {mse_rf:.4f}")
print(f"R² Score: {r2_rf:.4f}")
print(f"Mean Absolute Error: {mae_rf:.4f}\n")
```

```
Random Forest Regressor Performance:
Mean Squared Error: 502.6280
R² Score: -0.0134
Mean Absolute Error: 11.3791
```

6.4 XGBoost Regressor

```
import xgboost as xgb

# Initialize XGBoost Regressor
xgb_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', xgb.XGBRegressor(
        random_state=42,
        objective='reg:squarederror',
        device='cuda',  # Use GPU acceleration
        tree_method='hist',  # Use GPU acceleration
        #predictor='predictor', # Note necessary anymore
    ))
])

# Train and evaluate
mse_xgb, r2_xgb, mae_xgb, y_pred_xgb =
train_and_evaluate_model(xgb_pipeline)

print("XGBoost Regressor Performance:")
print(f"Mean Squared Error: {mse_xgb:.4f}")
print(f"R² Score: {r2_xgb:.4f}")
print(f"Mean Absolute Error: {mae_xgb:.4f}\n")
```

```
XGBoost Regressor Performance:
Mean Squared Error: 515.7646
R² Score: -0.0399
Mean Absolute Error: 11.3249


/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158:
UserWarning: [18:23:27] WARNING:
/workspace/src/common/error_msg.cc:58: Falling back to prediction
using DMatrix due to mismatched devices. This might lead to higher
memory usage and slower performance. XGBoost is running on: cuda:0,
```

```
while the input data is on: cpu.
Potential solutions:
- Use a data structure that matches the device ordinal in the booster.
- Set the device for booster before call to inplace_predict.

This warning will only be shown once.

  warnings.warn(smsg, UserWarning)
```

6.5 LightGBM Regressor

```
import lightgbm as lgb

# Initialize LightGBM Regressor
lgb_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', lgb.LGBMRegressor(
        random_state=42,
        n_jobs=-1,
        device='cpu'  # Use GPU
    ))
])

# Train and evaluate
mse_lgb, r2_lgb, mae_lgb, y_pred_lgb =
train_and_evaluate_model(lgb_pipeline)

print("LightGBM Regressor Performance:")
print(f"Mean Squared Error: {mse_lgb:.4f}")
print(f"R² Score: {r2_lgb:.4f}")
print(f"Mean Absolute Error: {mae_lgb:.4f}\n")
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead
of testing was 0.040867 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 22500
[LightGBM] [Info] Number of data points in the train set: 712474,
number of used features: 105
[LightGBM] [Info] Start training from score 0.579041
LightGBM Regressor Performance:
Mean Squared Error: 488.5778
R² Score: 0.0149
Mean Absolute Error: 10.9417
```

6.6 CatBoost Regressor

```
import catboost as cb
```

```
# Initialize CatBoost Regressor
cb_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', cb.CatBoostRegressor(
        random_state=42,
        verbose=0,
        task_type="GPU",  # Use GPU acceleration
        devices='0'
    ))
])

# Train and evaluate
mse_cb, r2_cb, mae_cb, y_pred_cb =
train_and_evaluate_model(cb_pipeline)

print("CatBoost Regressor Performance:")
print(f"Mean Squared Error: {mse_cb:.4f}")
print(f"R² Score: {r2_cb:.4f}")
print(f"Mean Absolute Error: {mae_cb:.4f}\n")
```

```
CatBoost Regressor Performance:
Mean Squared Error: 495.4529
R² Score: 0.0010
Mean Absolute Error: 11.0310
```

6.7 AdaBoost Regressor with Multiprocessing CPUs

```
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.pipeline import Pipeline

# Inizializza AdaBoost Regressor con un base_estimator ottimizzato per
multicore
ada_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', AdaBoostRegressor(
        estimator=DecisionTreeRegressor(
            random_state=42,
            max_depth=3
        ),
        n_estimators=100,    # Numero di alberi
        learning_rate=0.1,   # Tasso di apprendimento
        random_state=42
    ))
])

# Allena e valuta
mse_ada, r2_ada, mae_ada, y_pred_ada =
train_and_evaluate_model(ada_pipeline)
```

```
print("Prestazioni di AdaBoost Regressor (Multicore):")
print(f"Mean Squared Error: {mse_ada:.4f}")
print(f"R² Score: {r2_ada:.4f}")
print(f"Mean Absolute Error: {mae_ada:.4f}\n")
```

```
Prestazioni di AdaBoost Regressor (Multicore):
Mean Squared Error: 532.3386
R² Score: -0.0733
Mean Absolute Error: 11.8542
```

6.8 XGBoost Random Forest Regressor

```
# Set parameters for Random Forest with XGBoost
params = {
    "objective": "reg:squarederror",
    "learning_rate": 1,
    "max_depth": 5,
    "subsample": 0.8,
    "colsample_bynode": 0.8,
    "num_parallel_tree": 100,
    "tree_method": "hist",
    "random_state": 42,
    "device": "cuda",
    "predictor": "gpu_predictor"
}

# Initialize XGBoost Random Forest Regressor
xgb_rf_regressor = xgb.XGBRegressor(**params, n_estimators=1)

# Create the pipeline
xgb_rf_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', xgb_rf_regressor)
])

# Train and evaluate
mse_rfx, r2_rfx, mae_rfx, y_pred_rfx =
train_and_evaluate_model(xgb_rf_pipeline)

print("XGBoost Random Forest Regressor Performance:")
print(f"Mean Squared Error: {mse_rfx:.4f}")
print(f"R² Score: {r2_rfx:.4f}")
print(f"Mean Absolute Error: {mae_rfx:.4f}\n")
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158:
UserWarning: [19:31:31] WARNING: /workspace/src/learner.cc:740:
Parameters: { "predictor" } are not used.
```

```
  warnings.warn(smsg, UserWarning)

XGBoost Random Forest Regressor Performance:
Mean Squared Error: 493.1193
R² Score: 0.0057
Mean Absolute Error: 10.9972
```

7. Evaluating Model Performance

```
# Create a DataFrame to compare model performances
performance_df = pd.DataFrame({
    'Model': [
        'Linear Regression',
        'Decision Tree',
        'Random Forest',
        'XGBoost',
        'LightGBM',
        'CatBoost',
        'AdaBoost',
        'XGBoost RF'
    ],
    'Mean Squared Error': [
        mse_lr,
        mse_dt,
        mse_rf,
        mse_xgb,
        mse_lgb,
        mse_cb,
        mse_ada,
        mse_rfx
    ],
    'Mean Absolute Error': [
        mae_lr,
        mae_dt,
        mae_rf,
        mae_xgb,
        mae_lgb,
        mae_cb,
        mae_ada,
        mae_rfx
    ],
    'R² Score': [
        r2_lr,
        r2_dt,
        r2_rf,
        r2_xgb,
        r2_lgb,
```

```
        r2_cb,
        r2_ada,
        r2_rfx
    ]
})

print("Model Performance Comparison:")
performance_df
```

```
Model Performance Comparison:

                Model  Mean Squared Error  Mean Absolute Error  R²
Score
0  Linear Regression          526.174891            13.549635 -
0.060914
1      Decision Tree         1038.177294            17.118884 -
1.093252
2      Random Forest          502.627983            11.379111 -
0.013437
3            XGBoost          515.764605            11.324924 -
0.039924
4           LightGBM          488.577767            10.941750
0.014892
5           CatBoost          495.452879            11.030964
0.001030
6           AdaBoost          532.338609            11.854186 -
0.073342
7         XGBoost RF          493.119267            10.997198
0.005735
```

8. Hyperparameter Tuning with RandomizedSearchCV

8.1 XGBoost Random Forest Regressor Hyperparameter Tuning

```
from sklearn.model_selection import RandomizedSearchCV

# Define the pipeline with XGBRFRegressor
xgbrf_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', xgb.XGBRFRegressor(tree_method='hist',
device='cuda', random_state=42))
])

# Define the parameter grid for XGBRFRegressor
xgbrf_param_grid = {
    'regressor__n_estimators': [200, 400],
    'regressor__max_depth': [10, 20],
    'regressor__learning_rate': [0.01, 0.1],
    'regressor__subsample': [0.8, 1],
    'regressor__colsample_bynode': [0.8, 1]
```

```
}

# Initialize RandomizedSearchCV
xgbrf_random_search = RandomizedSearchCV(
    estimator=xgbrf_pipeline,
    param_distributions=xgbrf_param_grid,
    n_iter=10,
    cv=3,
    verbose=2,
    random_state=42,
    n_jobs=-1,
    scoring='neg_mean_squared_error'
)

from joblib import parallel_backend

with parallel_backend('threading'):
    xgbrf_random_search.fit(X_train, y_train)


# Best parameters
print("Best parameters found: ", xgbrf_random_search.best_params_)

# Best estimator
best_xgbrf = xgbrf_random_search.best_estimator_

# Predict on test set with best estimator
y_pred_best_xgbrf = best_xgbrf.predict(X_test)

# Evaluate the best model
mse_best_xgbrf = mean_squared_error(y_test, y_pred_best_xgbrf)
r2_best_xgbrf = r2_score(y_test, y_pred_best_xgbrf)
mae_best_xgbrf = mean_absolute_error(y_test, y_pred_best_xgbrf)

print("\nBest XGBoost Random Forest Regressor Performance After
Hyperparameter Tuning:")
print(f"Mean Squared Error: {mse_best_xgbrf:.4f}")
print(f"R² Score: {r2_best_xgbrf:.4f}")
print(f"Mean Absolute Error: {mae_best_xgbrf:.4f}")
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
```

```
Best parameters found:  {'regressor__subsample': 0.8,
'regressor__n_estimators': 200, 'regressor__max_depth': 10,
'regressor__learning_rate': 0.1, 'regressor__colsample_bynode': 0.8}

Best XGBoost Random Forest Regressor Performance After Hyperparameter
Tuning:
Mean Squared Error: 494.3506
R² Score: 0.0033
Mean Absolute Error: 11.0771
```

8.2 XGBoost Regressor Hyperparameter Tuning

```
# Define the parameter grid for XGBoost
xgb_param_grid = {
    'regressor__n_estimators': [100, 200],
    'regressor__max_depth': [3, 5, 7],
    'regressor__learning_rate': [0.01, 0.1],
    'regressor__subsample': [0.6, 0.8, 1.0],
    'regressor__colsample_bytree': [0.6, 0.8, 1.0],
    'regressor__gamma': [0, 0.1, 0.2],
}

# Initialize RandomizedSearchCV
xgb_random_search = RandomizedSearchCV(
    estimator=xgb_pipeline,
    param_distributions=xgb_param_grid,
    n_iter=10,
    cv=3,
    verbose=2,
    random_state=42,
    n_jobs=-1,
    scoring='neg_mean_squared_error'
)

# Fit RandomizedSearchCV
xgb_random_search.fit(X_train, y_train)

# Best parameters
print("Best parameters found for XGBoost: ",
xgb_random_search.best_params_)

# Best estimator
best_xgb = xgb_random_search.best_estimator_

# Predict on test set with best estimator
y_pred_best_xgb = best_xgb.predict(X_test)
```

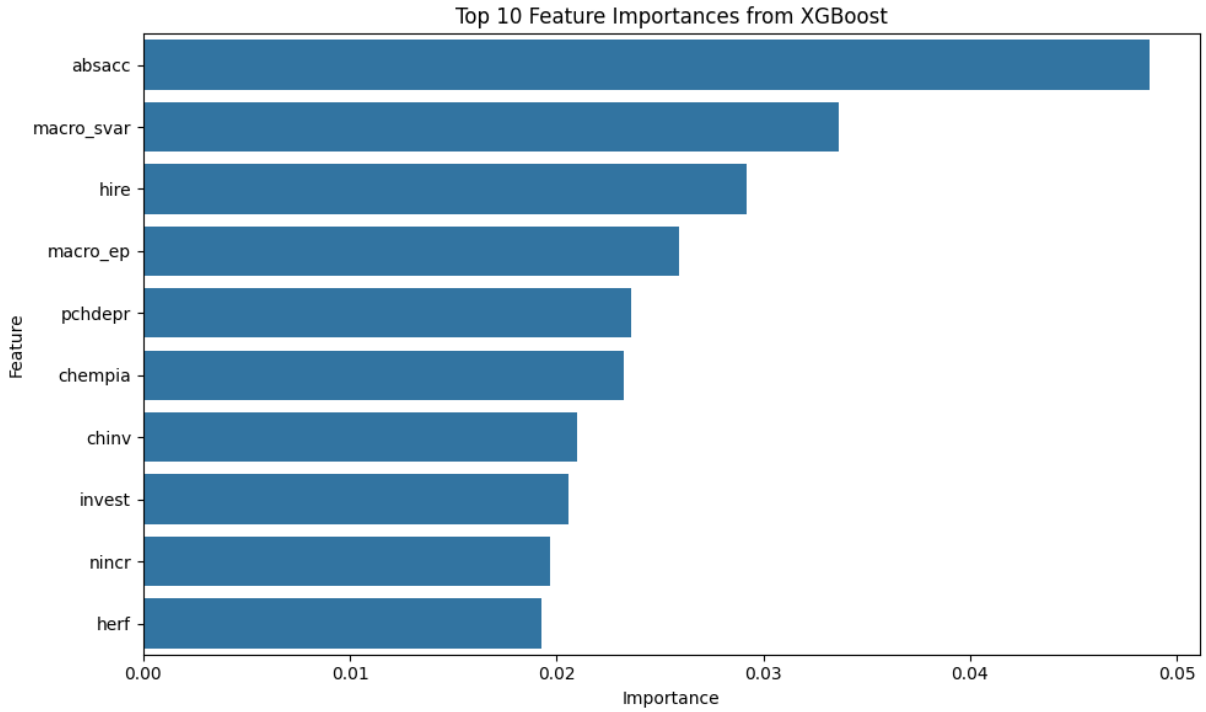
```python
# Evaluate the best model
mse_best_xgb = mean_squared_error(y_test, y_pred_best_xgb)
r2_best_xgb = r2_score(y_test, y_pred_best_xgb)
mae_best_xgb = mean_absolute_error(y_test, y_pred_best_xgb)

print("\nBest XGBoost Regressor Performance After Hyperparameter
Tuning:")
print(f"Mean Squared Error: {mse_best_xgb:.4f}")
print(f"R² Score: {r2_best_xgb:.4f}")
print(f"Mean Absolute Error: {mae_best_xgb:.4f}")
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits

/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158:
UserWarning: [19:48:32] WARNING:
/workspace/src/common/error_msg.cc:58: Falling back to prediction
using DMatrix due to mismatched devices. This might lead to higher
memory usage and slower performance. XGBoost is running on: cuda:0,
while the input data is on: cpu.
Potential solutions:
- Use a data structure that matches the device ordinal in the booster.
- Set the device for booster before call to inplace_predict.

This warning will only be shown once.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158:
UserWarning: [19:48:32] WARNING:
/workspace/src/common/error_msg.cc:58: Falling back to prediction
using DMatrix due to mismatched devices. This might lead to higher
memory usage and slower performance. XGBoost is running on: cuda:0,
while the input data is on: cpu.
Potential solutions:
- Use a data structure that matches the device ordinal in the booster.
- Set the device for booster before call to inplace_predict.

This warning will only be shown once.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158:
UserWarning: [19:48:36] WARNING:
/workspace/src/common/error_msg.cc:58: Falling back to prediction
using DMatrix due to mismatched devices. This might lead to higher
memory usage and slower performance. XGBoost is running on: cuda:0,
while the input data is on: cpu.
Potential solutions:
- Use a data structure that matches the device ordinal in the booster.
- Set the device for booster before call to inplace_predict.

This warning will only be shown once.
```

```
UserWarning: [19:48:49] WARNING:
/workspace/src/common/error_msg.cc:58: Falling back to prediction
using DMatrix due to mismatched devices. This might lead to higher
memory usage and slower performance. XGBoost is running on: cuda:0,
while the input data is on: cpu.
Potential solutions:
- Use a data structure that matches the device ordinal in the booster.
- Set the device for booster before call to inplace_predict.

This warning will only be shown once.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158:
UserWarning: [19:48:49] WARNING:
/workspace/src/common/error_msg.cc:58: Falling back to prediction
using DMatrix due to mismatched devices. This might lead to higher
memory usage and slower performance. XGBoost is running on: cuda:0,
while the input data is on: cpu.
Potential solutions:
- Use a data structure that matches the device ordinal in the booster.
- Set the device for booster before call to inplace_predict.

This warning will only be shown once.

  warnings.warn(smsg, UserWarning)
/usr/local/lib/python3.11/dist-packages/xgboost/core.py:158:
UserWarning: [19:48:49] WARNING:
/workspace/src/common/error_msg.cc:58: Falling back to prediction
using DMatrix due to mismatched devices. This might lead to higher
memory usage and slower performance. XGBoost is running on: cuda:0,
while the input data is on: cpu.
Potential solutions:
- Use a data structure that matches the device ordinal in the booster.
- Set the device for booster before call to inplace_predict.

This warning will only be shown once.

  warnings.warn(smsg, UserWarning)

Best parameters found for XGBoost:  {'regressor__subsample': 0.6,
'regressor__n_estimators': 200, 'regressor__max_depth': 5,
'regressor__learning_rate': 0.01, 'regressor__gamma': 0,
'regressor__colsample_bytree': 0.6}

Best XGBoost Regressor Performance After Hyperparameter Tuning:
Mean Squared Error: 485.4636
R² Score: 0.0212
Mean Absolute Error: 10.8227
```

9. Feature Importance Analysis

```
# Combine numerical and categorical feature names
feature_names = numerical_features + categorical_features

# Get feature importances
importances = best_xgb.named_steps['regressor'].feature_importances_

# Ensure the lengths match
if len(feature_names) != len(importances):
    raise ValueError("Length of feature names and importances must
match.")

# Create a DataFrame for visualization
feature_importances = pd.DataFrame({
    'Feature': feature_names,
    'Importance': importances
})

# Sort the DataFrame based on importance
feature_importances.sort_values(by='Importance', ascending=False,
inplace=True)

# Plot the top 10 features
plt.figure(figsize=(10, 6))
sns.barplot(x='Importance', y='Feature',
data=feature_importances.head(10))
plt.title('Top 10 Feature Importances from XGBoost')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```

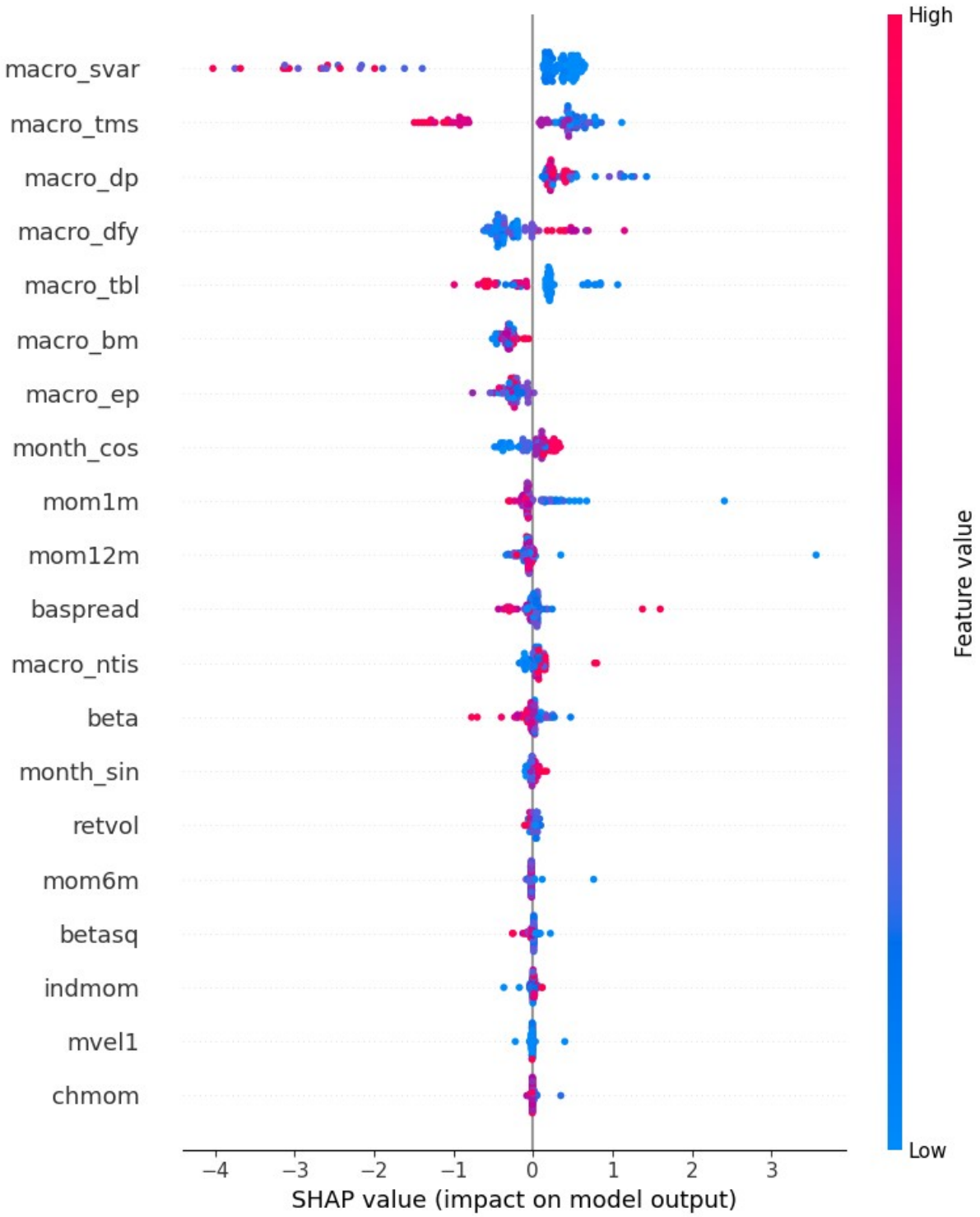


10. SHAP Values for Model Interpretation

```
# Load JS visualization code to render SHAP plots in notebooks
shap.initjs()
```

```
<IPython.core.display.HTML object>
```

```
# Prepare sample data for SHAP
X_sample = X_test.sample(100, random_state=42)  # Use a smaller sample
for efficiency
y_sample = y_test.loc[X_sample.index]

# Preprocess the sample data
X_sample_processed =
best_xgb.named_steps['preprocessor'].transform(X_sample)

# Ensure that X_sample_processed is a numpy array of float64
X_sample_processed = np.array(X_sample_processed, dtype=np.float64)

# Initialize the SHAP explainer
explainer = shap.Explainer(best_xgb.named_steps['regressor'])

# Calculate SHAP values
shap_values = explainer(X_sample_processed)

# Summary plot
shap.summary_plot(shap_values, features=X_sample_processed,
feature_names=feature_names)
```



11. Neural Network Implementation with GPU Acceleration

```
# Clear TensorFlow session
tf.keras.backend.clear_session()
```

```
# Apply preprocessing to the entire dataset
X_train_processed = preprocessor.fit_transform(X_train)
X_test_processed = preprocessor.transform(X_test)

# Convert to TensorFlow datasets for efficient loading and prefetching
batch_size = 256
train_dataset = tf.data.Dataset.from_tensor_slices((X_train_processed,
y_train))
train_dataset =
train_dataset.shuffle(buffer_size=1024).batch(batch_size).prefetch(tf.
data.experimental.AUTOTUNE)

test_dataset = tf.data.Dataset.from_tensor_slices((X_test_processed,
y_test))
test_dataset =
test_dataset.batch(batch_size).prefetch(tf.data.experimental.AUTOTUNE)

# Define the neural network architecture
def create_model():
    model = keras.Sequential([
        layers.Dense(256, activation='relu',
input_shape=(X_train_processed.shape[1],)),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(64, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(1)
    ])
    return model

model = create_model()

# Compile the model
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='mse',
    metrics=['mae']
)

# Define early stopping and model checkpoint
early_stop = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)
```

```
checkpoint_cb = keras.callbacks.ModelCheckpoint(
    "best_model.keras",
    save_best_only=True
)

# Train the model using GPU acceleration
history = model.fit(
    train_dataset,
    validation_data=test_dataset,
    epochs=100,
    callbacks=[early_stop, checkpoint_cb],
    verbose=1
)

# Load the best model
model = keras.models.load_model("best_model.keras")

# Evaluate the model
loss, mae = model.evaluate(test_dataset)
print(f"\nNeural Network Performance:")
print(f"Mean Squared Error: {loss:.4f}")
print(f"Mean Absolute Error: {mae:.4f}")

# Plot training & validation loss
plt.figure(figsize=(10,6))
plt.plot(history.history['loss'], label='Training Loss (MSE)')
plt.plot(history.history['val_loss'], label='Validation Loss (MSE)')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error')
plt.legend()
plt.show()
```

```
2024-11-28 19:48:59.411067: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1928] Created
device /job:localhost/replica:0/task:0/device:GPU:0 with 10728 MB
memory:  -> device: 0, name: NVIDIA GeForce RTX 4090, pci bus id:
0000:42:00.0, compute capability: 8.9
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py
:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to
a layer. When using Sequential models, prefer using an `Input(shape)`
object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Epoch 1/100

WARNING: All log messages before absl::InitializeLog() is called are
written to STDERR
I0000 00:00:1732823343.256117   15339 service.cc:145] XLA service
```

```
0x2c91eb40 initialized for platform CUDA (this does not guarantee that
XLA will be used). Devices:
I0000 00:00:1732823343.256219   15339 service.cc:153]   StreamExecutor
device (0): NVIDIA GeForce RTX 4090, Compute Capability 8.9
2024-11-28 19:49:03.310364: I
tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:268]
disabling MLIR crash reproducer, set env var
`MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
2024-11-28 19:49:03.551990: I
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:465] Loaded
cuDNN version 8906

  53/2784 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - loss: 666.1937 - mae:
16.7211

I0000 00:00:1732823348.284442   15339 device_compiler.h:188] Compiled
cluster using XLA!  This line is logged at most once for the lifetime
of the process.

2784/2784 ━━━━━━━━━━━━━━━━━━━━ 22s 5ms/step - loss: 352.0655 - mae:
10.6453 - val_loss: 491.2068 - val_mae: 10.8970
Epoch 2/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 351.0525 - mae:
10.5941 - val_loss: 497.5776 - val_mae: 11.0478
Epoch 3/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 350.3141 - mae:
10.5747 - val_loss: 500.9108 - val_mae: 11.1636
Epoch 4/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 349.6096 - mae:
10.5855 - val_loss: 496.2851 - val_mae: 11.0600
Epoch 5/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 349.5616 - mae:
10.5708 - val_loss: 512.8214 - val_mae: 11.3210
Epoch 6/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 347.2984 - mae:
10.5749 - val_loss: 501.8826 - val_mae: 11.3916
Epoch 7/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 346.0555 - mae:
10.5702 - val_loss: 508.7905 - val_mae: 11.5145
Epoch 8/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 345.4243 - mae:
10.5521 - val_loss: 511.3690 - val_mae: 11.5944
Epoch 9/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 344.7937 - mae:
10.5552 - val_loss: 510.1950 - val_mae: 11.5976
Epoch 10/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 342.3572 - mae:
10.5438 - val_loss: 508.4531 - val_mae: 11.8406
Epoch 11/100
2784/2784 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - loss: 341.8976 - mae:
```

```
10.5339 - val_loss: 520.3240 - val_mae: 11.7807
696/696 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 436.9028 - mae:
10.7192

Neural Network Performance:
Mean Squared Error: 491.2068
Mean Absolute Error: 10.8970
```