International Institute of Information Technology Bangalore

ML Project Report: Medical Equipments Cost Prediction (Checkpoint 1)

Team Name: Predictify

Team Members

1. Kartikeya Dimri - IMT2023126
2. Ayush Mishra - IMT2023129
3. Harsh Sinha - IMT2023571

Github Repo Link:

https://github.com/kartikeya-dimri/Medical-Equipment-Cost-Prediction

Contents

1 Task

Training ML models to predict the transport cost of medical equipment orders is essentially a **regression problem**, where the target variable is the cost of transporting each order. Several factors, as mentioned below, contribute to predicting the final transport cost, making this task a matter of understanding complex relationships between equipment characteristics, delivery requirements, and logistical constraints to enable better planning and fair pricing.

2 Background

Transporting sensitive medical equipment comes with unique challenges:

- Ensuring fragile devices arrive safely.
- Handling urgent shipments for critical hospital needs.
- Managing cross-border deliveries and rural hospital locations.
- Coordinating installation and setup services when required.

Your company specializes in delivering medical equipment sourced from multiple suppliers worldwide. The goal is to build a model that predicts the transport cost of each order, enabling better logistics planning and fair pricing.

3 Dataset Description

The dataset contains information about medical equipment transport orders from various suppliers to hospitals worldwide. It provides an opportunity to tackle a realistic logistics cost prediction problem — one that mirrors the complexity of actual supply-chain operations in the healthcare sector.

With 5,000 records and 20 columns, the dataset captures a wide range of details such as supplier reliability, equipment dimensions, shipping modes, urgency levels, and whether installation services were required. Each row represents a unique delivery order, combining both numerical and categorical variables.

This dataset not only tests a model's ability to handle mixed data types but also requires thoughtful feature engineering and preprocessing. The challenge lies in understanding how multiple logistical factors — like fragility, rural locations, or cross-border deliveries — collectively influence the overall transport cost. Ultimately, the dataset serves as a realistic foundation for developing, comparing, and fine-tuning regression models aimed at optimizing pricing and operational efficiency in medical logistics.

4 Column Descriptions

The columns provided in the dataset are as follows:

Table 1: Dataset Column Descriptions

No.	Column Name	Description	Type
1	Hospital_Id	Unique identification number of the hospital	Categorical
2	Supplier_Name	Name of the medical equipment supplier	Categorical
3	Supplier_Reliability	Supplier's reputation score in the market (higher means more reliable)	Numerical
4	Equipment_Height	Height of the equipment	Numerical
5	Equipment_Width	Width of the equipment	Numerical
6	Equipment_Weight	Weight of the equipment	Numerical
7	Equipment_Type	The type of equipment (e.g., diagnostic device, surgical tool)	Categorical
8	Equipment_Value	Monetary value of the equipment	Numerical
9	Base_Transport_Fee	Base fee for transporting the equipment	Numerical
10	CrossBorder_Shipping	Whether the delivery is international	Categorical (Boolean)
11	Urgent_Shipping	Whether the delivery was in express/urgent mode	Categorical (Boolean)
12	Installation_Service	Whether installation/setup service was included	Categorical (Boolean)
13	Transport_Method	Mode of transport used (e.g., air, road, sea)	Categorical
14	Fragile_Equipment	Whether the equipment is fragile	Categorical (Boolean)
15	Hospital_Info	Additional information about the hospital	Categorical
16	Rural_Hospital	Whether the hospital is located in a remote/rural area	Categorical (Boolean)
17	Order_Placed_Date	Date when the order was placed	Categorical (Date)
18	Delivery_Date	Date when the delivery occurred	Categorical (Date)
19	Hospital_Location	Location of the hospital	Categorical
20	Transport_Cost	Target variable — cost of transporting the order	Numerical

5 EDA and Preprocessing

In this phase, we performed an in-depth Exploratory Data Analysis (EDA) to understand data structure, detect patterns, correlations, and anomalies, and assess feature relationships with the target variable. Based on these insights, we applied targeted preprocessing — cleaning invalid records, engineering new features, handling skewness, splitting data, and building robust pipelines to ensure reliable model training.

5.1 Import Libraries and Load Dataset

In this script, we imported pandas and numpy for efficient data manipulation, and matplotlib, seaborn, and missingno for visualizing distributions, correlations, and missing values. From scipy, zscore was used for outlier detection, while sklearn provided tools for preprocessing, pipeline creation, train-test splitting, and handling numeric and categorical transformations through Pipeline and ColumnTransformer, enabling robust, model-ready datasets.

5.2 Data Loading and Basic Cleaning

The following steps were performed to load the dataset and prepare it for further analysis:

- **Load Dataset:** Read the CSV file into a pandas DataFrame, display the top rows, and check the initial shape.
- **Handle Missing Values:** Clean string/object columns by stripping spaces and replacing blanks or invalid strings with `NaN`.
- **Normalize Yes/No Columns:** Standardize categorical columns like `CrossBorder_Shipping` and `Urgent_Shipping` to consistent "Yes"/"No" values.
- **Convert Date Columns:** Convert order and delivery dates into proper datetime format for accurate calculations.
- **Drop Duplicates and Quick Check:** Remove exact duplicate rows, check dataset shape after cleaning, and examine missing value counts (both raw and percentage).

5.3 Exploratory Data Analysis (EDA)

In this phase, we performed a detailed Exploratory Data Analysis (EDA) to understand the structure of the dataset, detect patterns, and uncover potential anomalies. The following steps summarize the approach:

1. **Identification of Feature Types:**
   We started by identifying the types of features in the dataset:
   - **Numeric Features:** Columns with continuous or discrete numeric values (excluding the target and date-derived features).
   - **Categorical Features:** Columns with string or boolean values representing categories.
   - **Date-Derived Features:** Columns derived from order and delivery dates, such as delivery days, order month, and day of the week.

   This classification helped us apply appropriate analysis techniques for each type of feature.

2. **Target Variable Analysis (Transport_Cost):**
   Understanding the distribution of the target variable is crucial for regression tasks:
   - We plotted the **original distribution** of `Transport_Cost` to check for skewness, outliers, or unusual patterns.
   - We also applied a **log-transformation** (log(Transport_Cost + 1)) to reduce skewness and better approximate a normal distribution, which can improve model performance.

- Skewness statistics were calculated for both original and log-transformed distributions to quantitatively assess asymmetry.

3. **Creation of Date-Derived Features for Better Understanding:**
   To gain more insight into temporal patterns in the data, we created additional features from the order and delivery dates. These features are intended purely for data exploration and understanding, and may or may not be used in subsequent modeling:

   - **Delivery_Days:** Calculated as the difference in days between `Delivery_Date` and `Order_Placed_Date`, giving the actual delivery duration.
   - **Order_Month:** Extracted the month from the order date to analyze seasonal trends.
   - **Order_Day_of_Week:** Extracted the day of the week from the order date to detect weekday patterns.
   - **Order_Is_Weekend:** A boolean indicator for whether the order was placed on a weekend, helping to understand weekend vs. weekday behavior.

   These features help us better understand patterns in delivery durations and temporal effects across the dataset, without implying they will necessarily be included in the final model.

4. **Numeric Feature Analysis:**
   For numeric features, we performed the following steps:

   - Computed **descriptive statistics** such as mean, standard deviation, minimum, maximum, and quartiles to understand feature distributions.
   - Calculated **skewness** to detect non-normal distributions that may require transformation or scaling.
   - Visualized distributions using **histograms** and **boxplots** to identify outliers and understand spread and central tendency.

5. **Correlation Analysis:**
   Correlation analysis helps identify linear relationships between features and with the target:

   - We computed the **correlation matrix** for numeric features including the target variable.
   - A **heatmap** was plotted to visually highlight strong positive or negative correlations.
   - Insights from correlations guide feature selection and engineering, as highly correlated features can inform modeling strategies.

6. **Categorical Feature Analysis:**
   For categorical variables, the following analysis was performed:

   - Displayed the **frequency counts** for each category to understand the distribution of values.
   - Plotted **countplots** for low-cardinality features to visualize the frequency of each category.
   - For high-cardinality features (more than 20 unique values), we summarized the **top 10 categories** to avoid cluttered visualizations while still understanding major trends.

7. **Bivariate Analysis (Features vs Target):**
   To understand the relationship between features and the target variable:

   - For numeric features, **scatter plots** were used to visualize trends and detect potential linear or non-linear relationships with the target.
   - For categorical features, **boxplots** were plotted to observe the distribution of the target variable across different categories.
   - Date-derived features such as order month, day of the week, and weekend indicators were also analyzed against the target to identify potential temporal patterns.

8. **Outlier Detection:**
   Outliers can significantly affect regression models. We applied the following steps:

- Computed **Z-scores** for numeric features and the target variable to detect extreme values (Z-score > 3).
- Counted and summarized the number of outliers per column to identify features with significant anomalies.
- This analysis informs decisions for handling outliers during preprocessing and feature engineering.

9. **Investigation of Negative Duration (Delivery_Days):**
   Negative values in delivery duration indicate potential data entry errors:
   - Counted the number of rows where `Delivery_Days` is negative.
   - Analyzed the most common negative values and basic statistics (min, max, mean) to understand the extent of the issue.
   - This helps in deciding whether to correct, remove, or treat these anomalous rows during preprocessing.
10. **Investigation of Negative Transport Cost:**
    Negative transport costs are logically inconsistent and may indicate errors:
    - Counted the number of rows with negative `Transport_Cost`.
    - Reviewed the top negative values and descriptive statistics to quantify the anomaly.
    - Understanding these anomalies helps decide on cleaning or imputing strategies.
11. **Investigation of Equipment Weight vs Value:**
    Feature relationships can reveal inconsistencies or missing data:
    - Calculated correlation between `Equipment_Weight` and `Equipment_Value`.
    - Checked for zero or missing values in both columns and reported the percentage of affected rows.
    - This step identifies potential data quality issues and informs feature handling.
12. **Investigation of Equipment Height vs Width:**
    Similar to weight-value analysis, dimensional consistency is important:
    - Calculated correlation between `Equipment_Height` and `Equipment_Width`.
    - Checked for zero or missing values and reported the percentage of affected rows.
    - These insights can guide feature engineering and data correction strategies.

Overall, this comprehensive EDA provided a clear understanding of the dataset, highlighted potential data issues (negative durations, negative costs, missing or zero values), and guided subsequent preprocessing and feature engineering steps. For full details, plots, and code, see the EDA file here.

6 Preprocessing Strategies

Following the insights gained from the Exploratory Data Analysis (EDA), several preprocessing pipelines were developed and tested to prepare the data for various modeling approaches. Each pipeline focused on different feature selection, encoding, and imputation strategies. The goal was to evaluate how these different preprocessing choices impact model performance.

Below is a summary of the key steps performed in each preprocessing script:

6.1 `preprocessing.py` (Initial Full Pipeline)

This script represented the initial comprehensive preprocessing approach:

- **Target Handling:** Clipped negative `Transport_Cost` to 0 and applied a log(1+x) transformation.
- **Feature Engineering:** Calculated `Delivery_Time_Days` after converting date columns.

- **Imputation:** Used median for missing numerical features and the most frequent value for missing categorical features.
- **Encoding:** Applied One-Hot Encoding (OHE) to all categorical features.
- **Scaling:** Applied `StandardScaler` to all numerical features.
- **Dropped Columns:** Removed `Hospital_Id`, `Supplier_Name`, `Hospital_Location`, and date columns.
- **Output:** Saved processed data as NumPy arrays (`X_train_processed.npy`, etc.).

6.2 `preprocessing2.py` (Date Flipping & Median Imputation)

This script built upon the first by adding specific data cleaning steps identified in EDA:

- **Date Flipping:** Corrected instances where `Delivery_Date` was before `Order_Placed_Date` by swapping them before calculating `Delivery_Time_Days`.
- **Target Handling:** Same as `preprocessing.py`.
- **Imputation:** Same as `preprocessing.py` (median for numerical, most frequent for categorical).
- **Encoding & Scaling:** Same as `preprocessing.py`.
- **Dropped Columns:** Same as `preprocessing.py`.
- **Output:** Saved processed data with a `2` suffix (e.g., `X_train_processed2.npy`).

6.3 `preprocessing3.py` (Minimal Feature Set)

This script tested a drastically simplified feature set based on EDA correlation analysis:

- **Feature Selection:** Kept ONLY `Equipment_Weight` and `Equipment_Value`. All other features were dropped.
- **Target Handling:** Same as `preprocessing.py`.
- **Imputation:** Used median for missing values in the two selected features.
- **Scaling:** Applied `StandardScaler` to the two selected features.
- **Output:** Saved processed data with a `3` suffix (e.g., `X_train_processed3.npy`).

6.4 `preprocessing3_no_outliers.py` (Minimal Features + Outlier Removal)

This script extended the minimal feature set by attempting to handle outliers:

- **Feature Selection:** Kept ONLY `Equipment_Weight` and `Equipment_Value`.
- **Outlier Handling (IQR):**
  - Calculated IQR bounds for `Equipment_Weight` and `Equipment_Value` using the training data.
  - **Dropped** rows from the training set where values fell outside 1.5 * IQR.
  - **Capped** values in the test set at the calculated training bounds.
- **Target Handling:** Same as `preprocessing.py`, but aligned with the outlier-removed training data.
- **Imputation & Scaling:** Applied median imputation and scaling to the remaining/capped data.
- **Output:** Saved processed data with a `3_no_outliers` suffix.

6.5 `preprocessing4.py` (Advanced Feature Engineering)

This script implemented more sophisticated feature engineering based on EDA:

- **Target Handling & Date Flipping:** Same as `preprocessing2.py`.
- **Feature Engineering:**
  - Extracted `State` and `Is_Military_Address` from `Hospital_Location`.
  - Extracted `Order_Month` and `Order_Year` from dates.
  - Created `Value_Per_Weight` interaction feature (imputing weight/value beforehand).
  - Manually mapped binary ('Yes'/'No') and ordinal (`Hospital_Info`) features to numerical representations (0/1 or 0/1/2).
- **Imputation:** Median for numerical, most frequent for remaining categorical text features (`Equipment_Type`, `Transport_Method`, `State`).
- **Encoding & Scaling:** OHE for text categoricals, `StandardScaler` for numericals.
- **Dropped Columns:** Dropped original location, dates, and `Supplier_Name`.
- **Output:** Saved processed data with a `4` suffix.

6.6 `preprocessing_add_<feature>.py` (Base + Single Feature)

This series of six scripts tested the impact of adding one feature back to the minimal `preprocessing3.py` set:

- **Base Features:** Started with `Equipment_Weight` and `Equipment_Value`.
- **Added Feature:** Each script added one of the following: `Base_Transport_Fee`, `Supplier_Reliability`, `Delivery_Time_Days` (engineered), `Transport_Method`, `Urgent_Shipping` (mapped 0/1), or `Equipment_Type`.
- **Target Handling:** Same as `preprocessing.py`.
- **Imputation:** Median for numerical features, most frequent for categorical features.
- **Encoding & Scaling:** OHE for categorical (if present), `StandardScaler` for numerical.
- **Output:** Saved processed data with a suffix indicating the added feature (e.g., `X_train_processed_SupplierReliabi`

6.7 `catboost_preprocessing.py` (Native Categorical Handling)

This script was specifically designed for models like CatBoost that handle categorical features internally:

- **Target Handling & Date Flipping:** Same as `preprocessing2.py`.
- **Feature Engineering:** Calculated `Delivery_Time_Days`.
- **Imputation:** Used median for missing numerical features and the string `"Missing"` for missing categorical features.
- **Encoding: No One-Hot Encoding** was performed on categorical features.
- **Scaling:** Applied `StandardScaler` to numerical features.
- **Dropped Columns:** Dropped `Supplier_Name`, `Hospital_Location`, and date columns.
- **Output:** Saved processed data as CSV files (`catboost_train.csv`, `catboost_test.csv`) and saved the list of categorical feature names.

These varied approaches allowed us to systematically test the influence of different feature sets and preprocessing techniques on the predictive accuracy of various regression models. The results from models trained on these datasets are discussed in the following sections.

7 Model Training (Total 37 files)

Based on the various preprocessing pipelines developed, a range of regression models were trained and evaluated. The primary goal was to compare the performance of different algorithms and identify the impact of feature selection and preprocessing choices on predictive accuracy. Cross-validation and hyperparameter tuning were key components of this process.

7.1 Linear Regression

Explanation: This is the simplest regression model. It tries to find the best straight line (or hyperplane in higher dimensions) that fits the data points. It assumes a linear relationship between the input features and the target variable. It's fast and easy to interpret but can be too simple for complex relationships.

Files:

- `1_linear_regression.py` (Output: `output/linear_regression.csv`)
- `1_linear_regression_2.py` (Output: `output/linear_regression_2.csv`)
- `1_linear_regression_3.py` (Output: `output/linear_regression_3.csv`)

7.2 Polynomial Regression

Explanation: This extends linear regression by adding polynomial features (like squared or cubed terms of the original features). This allows the model to fit curves instead of just straight lines, capturing non-linear relationships. We used early stopping to find the best degree.

Files:

- `2_polynomial_regression.py` (Output: `output/polynomial_regression_early_stop.csv`)

7.3 Ridge Regression (L2 Regularization)

Explanation: A variation of linear regression that adds a penalty based on the *squared* size of the coefficients (weights). This helps prevent overfitting, especially when features are correlated, by shrinking the coefficients towards zero but rarely making them exactly zero. It keeps all features but reduces their impact.

Files:

- `3_ridge.py` (Output: `output/ridge_regression_tuned.csv`)
- `3_ridge_2.py` (Output: `output/ridge_tuned_2.csv`)
- `3_ridge_3.py` (Output: `output/ridge_tuned_3.csv`)

7.4 Lasso Regression (L1 Regularization)

Explanation: Another variation of linear regression that adds a penalty based on the *absolute* size of the coefficients. This not only prevents overfitting but also performs automatic **feature selection** by shrinking some coefficients exactly to zero, effectively removing those features from the model.

Files:

- `4_lasso.py` (Output: `output/lasso_regression_tuned.csv`)
- `4_lasso_2.py` (Output: `output/lasso_tuned_2.csv`)
- `4_lasso_3.py` (Output: `output/lasso_tuned_3.csv`)

7.5 ElasticNet Regression

Explanation: A combination of Ridge and Lasso. It includes both L1 and L2 penalties, controlled by an `l1_ratio` parameter. It aims to get the best of both worlds: handling correlated features like Ridge and performing feature selection like Lasso.

Files:

- 5_elastic_net.py (Output: output/elastic_net_tuned.csv)
- 5_elastic_net_2.py (Output: output/elastic_net_tuned_2.csv)
- 5_elastic_net_3.py (Output: output/elastic_net_tuned_3.csv)

7.6 Decision Tree Regression

Explanation: This model builds a tree-like structure where each internal node represents a test on a feature (e.g., "Is Weight > 500?"), and each leaf node contains a prediction (the average target value of samples reaching that leaf). It can capture complex non-linear patterns but is prone to overfitting if not pruned (limited in depth or leaf size).

Files:

- 8_decision_tree.py (Output: output/decision_tree_tuned_1.csv)
- 9_decision_tree_2.py (Output: output/decision_tree_tuned_2.csv)
- 10_decision_tree_3.py (Output: output/decision_tree_tuned_3.csv)

7.7 Random Forest Regression

Explanation: An ensemble method that builds many Decision Trees on different random subsets of the data and features ("forest"). The final prediction is the average of the predictions from all individual trees. This significantly reduces overfitting compared to a single Decision Tree and often results in high accuracy.

Files:

- 11_random_forest_2.py (Output: output/random_forest_tuned_2.csv)
- 11_random_forest_3.py (Output: output/random_forest_tuned_3.csv)
- 11_random_forest_4.py (Output: output/random_forest_tuned_4.csv)
- 14_three_feature_random_forest.py (Output: output/random_forest_best_single_feature_{suffix}.
- 18_random_forest_tuned_3_no_outliers.py (Output: output/random_forest_tuned_3_no_outliers.csv
- 20_random_forest_3_mae.py (Output: output/random_forest_grid_3_mae.csv)
- 21_random_forest_3_r2.py (Output: output/random_forest_grid_3_r2.csv)

7.8 Gradient Boosting Regression (Scikit-learn)

Explanation: An ensemble method that builds trees sequentially. Each new tree tries to correct the errors made by the previous ensemble of trees. It typically uses shallow trees and learns slowly (small learning rate) to achieve high accuracy but can be prone to overfitting if not tuned carefully. →

Files:

- 12_gradient_boost_2.py (Output: output/gradient_boosting_tuned_2.csv)
- 12_gradient_boost_3.py (Output: output/gradient_boosting_tuned_3.csv)
- 12_gradient_boost_optuna.py (Output: output/gradient_boosting_optuna_3.csv)
- 22_gradient_boost_3_mae.py (Output: output/gradient_boosting_grid_3_mae.csv)
- 22_gradient_boost_3_r2.py (Output: output/gradient_boosting_grid_3_r2.csv)

7.9 XGBoost Regression (Extreme Gradient Boosting)

Explanation: A highly optimized and popular implementation of gradient boosting. It includes advanced features like regularization (L1/L2), efficient tree building, handling of missing values, and parallel processing, often leading to better performance and speed compared to scikit-learn's basic Gradient Boosting.

Files:

- 13_xg_boost_2.py (Output: output/xgboost_grid_2.csv)
- 13_xg_boost_3.py (Output: output/xgboost_grid_3.csv)
- 19_xgboost_optuna.py (Output: output/xgboost_optuna_3.csv)

7.10 LightGBM Regression (Light Gradient Boosting Machine)

Explanation: Another fast, high-performance gradient boosting framework developed by Microsoft. It uses a different tree-building strategy (leaf-wise growth) compared to XGBoost (level-wise growth), which can be faster and more memory-efficient, especially on large datasets.

Files:

- `15_lightGDB.py` (Output: `output/lightgbm_refined_3.csv`)
- `17_lightgbm_optuna_SupplierReliability.py` (Output: `output/lightgbm_optuna_SupplierReliability`

7.11 CatBoost Regression

Explanation: A gradient boosting library developed by Yandex. Its main strength is its sophisticated, built-in handling of categorical features, often outperforming other models when many categorical variables are present. It uses techniques like ordered boosting and oblivious trees to improve accuracy and reduce overfitting.

Files:

- `6_catboost.py` (Output: `output/catboost_native_tuned.csv`)
- `6_catboost_2.py` (Output: `output/catboost_tuned_2.csv`)
- `7_catboost_optuna.py` (Output: `output/catboost_optuna_tuned.csv`)

7.12 Ensemble Model

Explanation: Combining predictions from multiple diverse, well-performing models can often lead to improved robustness and accuracy compared to any single model. Simple averaging is a common technique.

Files:

- `16_ensemble.py` (Output: `output/ensemble.csv`)

Evaluation and Tuning: For robust evaluation, **k-fold cross-validation** (typically with k=3 or k=5) was used during the training phase. To optimize performance, **hyperparameter tuning** was performed for most models using either **GridSearchCV** or **Optuna**. Different evaluation metrics were explored during tuning, including Root Mean Squared Error (RMSE) on the log-transformed target (equivalent to RMSLE), Mean Absolute Error (MAE), and R-squared (R^2).
Each model type was tested on multiple versions of the preprocessed data to understand the impact of feature engineering and selection. The final predictions for each variant were saved to separate CSV files. The selection of the best overall approach is based on cross-validation scores and performance on the Kaggle public leaderboard, as discussed in the next section.

8 Observations

- **Impact of Feature Set:** Models trained on the minimal feature set (`preprocessing3.py`, containing only 'Equipment_Weight' and 'Equipment_Value') consistently performed very well, often outperforming models trained on datasets with many more features. This suggests that these two features capture the majority of the predictive signal for transport cost.
- **Noise vs. Signal:** Adding more features, even with advanced engineering (`preprocessing4.py`) or careful selection (`preprocessing_add_<feature>.py` series), generally did not lead to significantly better cross-validation scores and sometimes resulted in worse performance. This indicates that many additional features might be acting as noise, potentially causing models like Random Forest and Gradient Boosting to overfit.
- **Benefit of Specific Features:** The systematic "add-one-feature" experiment (`14_three_feature_random_forest.py`) revealed that adding 'Supplier_Reliability' to the minimal feature set yielded the best improvement for the Random Forest model during cross-validation, achieving an RMSE of approximately 2.0723. This suggests 'Supplier_Reliability' offers valuable information complementary to weight and value.

- **Outlier Handling:** Removing outliers using the IQR method (`preprocessing3_no_outliers.py`) was tested (`18_random_forest_tuned_3_no_outliers.py`). Initial results indicate this did not improve performance compared to models trained on the original 'preprocessing3.py' data, suggesting the log transformation and robust models handle extreme values reasonably well.
- **Tuning Metric Influence:** Optimizing hyperparameters using different metrics (RMSE, MAE, R^2) led to different best parameter sets. Notably, optimizing Random Forest using R^2 (`21_random_forest_3_r2.py`) yielded the best performance on the Kaggle public leaderboard at the time of reporting, achieving **[4077502993.003]**, even though other metrics might have shown slightly better cross-validation scores. This highlights the importance of aligning the tuning metric with the final evaluation criteria when possible, or suggests potential differences between the cross-validation setup and the Kaggle test set.
- **Model Comparison:** Among the tested algorithms on the best-performing dataset (`preprocessing3.py`), ensemble methods like Random Forest and Gradient Boosting consistently outperformed other models after hyperparameter tuning. Other gradient boosting variants (LightGBM, XGBoost, CatBoost) also showed competitive performance.

9 Results

The best-performing model identified during the experimentation phase was the **Random Forest Regressor**, trained on the minimal `preprocessing3.py` dataset using only two selected features and optimized with the R^2 metric. The model achieved the **highest Kaggle public leaderboard score of 4,077,502,993.003**.

The hyperparameter search was conducted using the following parameter grid:

```
param_grid = {
    'n_estimators': [300, 400],        # Number of trees in the forest
    'max_depth': [5, 10, None],        # Maximum depth of the tree
    'min_samples_split': [2, 4, 6],    # Minimum samples required to split a node
    'min_samples_leaf': [1, 2, 3, 4]   # Minimum samples required at a leaf node
}
```

Among the tested configurations, the best model configuration corresponded to the parameters yielding the top leaderboard performance.

10 Interpretation: Why the Random Forest Model with Two Features Performed Best

One of the most striking outcomes from our experiments was that the **Random Forest** model trained on the simplified dataset from `preprocessing3.py`—containing only `Equipment_Weight` and `Equipment_Value`—and tuned using the R^2 metric (`21_random_forest_3_r2.py`) achieved the highest score on the Kaggle public leaderboard. Although this result may initially appear counterintuitive compared to models trained on richer feature sets, it offers several meaningful insights into the data and modeling process.

- **Dominance of Key Features (Signal Over Noise):** The outstanding performance suggests that `Equipment_Weight` and `Equipment_Value` are the primary factors influencing `Transport_Cost`. Logically, transportation costs are closely tied to how heavy an item is and its value, which affects insurance, handling, and risk. Other variables likely added more noise than useful information, reducing the overall predictive quality.
- **Overfitting in Larger Feature Sets:** Models trained on datasets with many additional features (such as those from `preprocessing2.py` or `preprocessing4.py`) tended to overfit. Complex ensemble methods like Random Forest and Gradient Boosting can easily capture spurious patterns when exposed to irrelevant or weakly correlated features. As a result, they performed well on training data but failed to generalize to unseen Kaggle test data.
- **Simplicity and Generalization:** Restricting the model to only the two most influential features forced it to focus on the fundamental relationship between weight, value, and cost. This simplicity helped the model generalize better and avoid overfitting, leading to improved performance on the leaderboard.

- **Effect of the Optimization Metric (R^2):** Unlike RMSE or MAE, which directly minimize prediction errors, optimizing for R^2 encourages the model to maximize the proportion of variance explained in the target variable. For this dataset, tuning based on R^2 may have helped the model capture broader trends in `Transport_Cost`, even if it did not achieve the lowest absolute error, resulting in better leaderboard performance.

References

[1] "15 Types of Regression Models in Machine Learning: Types & Examples." Pickl AI Blog, https://www.pickl.ai/blog/regression-in-machine-learning-types-examples/.

[2] "CatBoost — open-source gradient boosting on decision trees." CatBoost.ai, https://catboost.ai/.

[3] "MSE vs RMSE vs MAE vs MAPE vs R-Squared: When to Use?" VitalFlux, https://vitalflux.com/mse-vs-rmse-vs-mae-vs-mape-vs-r-squared-when-to-use/.