CSE228: Project Report

Predict the Item Price

Submitted to:
Dr. Mariam El Berri
Eng. Mohamed Essam

Submitted by:
Belal Anas Mohamed 23P0193
Omar Tamer 23P0174
Sara Ahmed 23P0184
Adam Tamer 23P0001
Yehya Mohamed 23P0067
Omar Elkhadragy 23P0085

Date of Submission:
28/12/2024

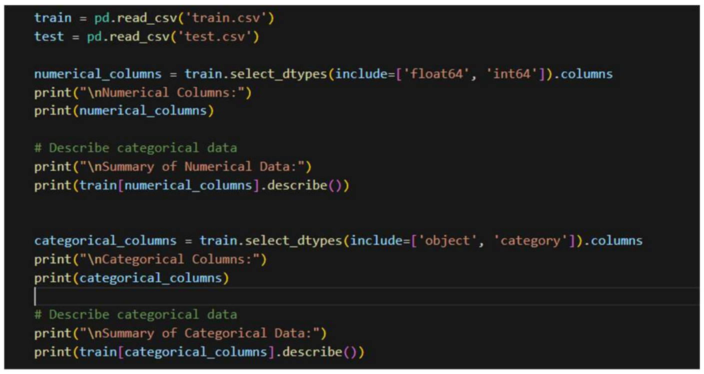
Contents

Abstract

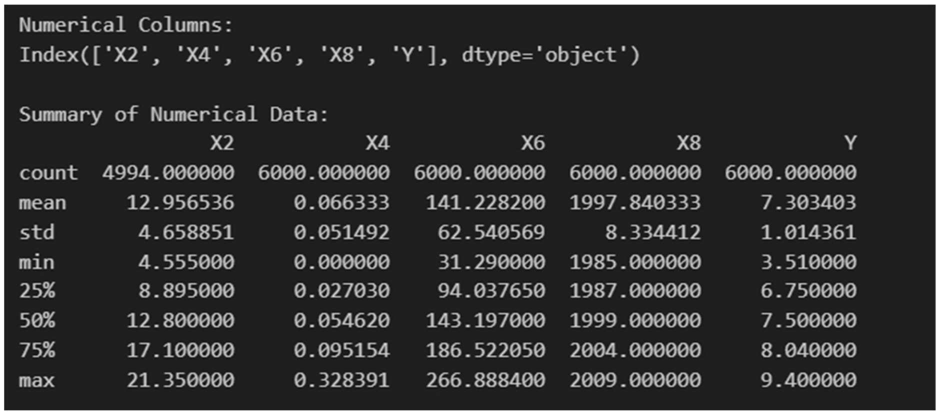
This project develops an AI system to predict product prices using machine learning models like Polynomial Regression, Support Vector Machines (SVM), XGBoost, and Random Forest. We cleaned and prepared the data to ensure reliable results.

1. DATA ANALYSIS

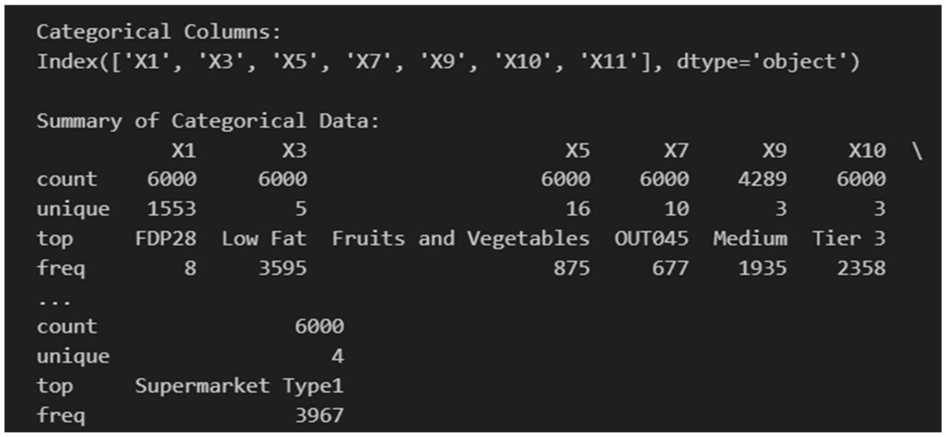
- **Describe Data**



```
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')

numerical_columns = train.select_dtypes(include=['float64', 'int64']).columns
print("\nNumerical Columns:")
print(numerical_columns)

# Describe categorical data
print("\nSummary of Numerical Data:")
print(train[numerical_columns].describe())


categorical_columns = train.select_dtypes(include=['object', 'category']).columns
print("\nCategorical Columns:")
print(categorical_columns)

# Describe categorical data
print("\nSummary of Categorical Data:")
print(train[categorical_columns].describe())
```

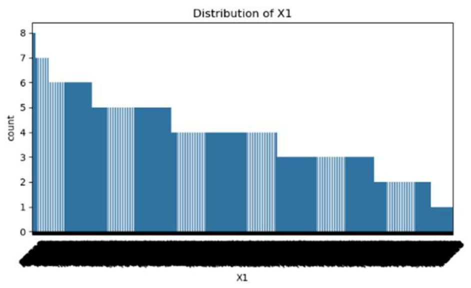


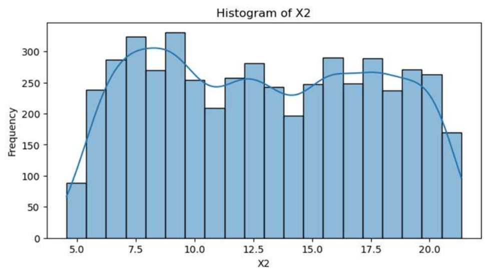
```
Numerical Columns:
Index(['X2', 'X4', 'X6', 'X8', 'Y'], dtype='object')

Summary of Numerical Data:
                X2           X4           X6           X8            Y
count  4994.000000  6000.000000  6000.000000  6000.000000  6000.000000
mean     12.956536     0.066333   141.228200  1997.840333     7.303403
std       4.658851     0.051492    62.540569     8.334412     1.014361
min       4.555000     0.000000    31.290000  1985.000000     3.510000
25%       8.895000     0.027030    94.037650  1987.000000     6.750000
50%      12.800000     0.054620   143.197000  1999.000000     7.500000
75%      17.100000     0.095154   186.522050  2004.000000     8.040000
max      21.350000     0.328391   266.888400  2009.000000     9.400000
```

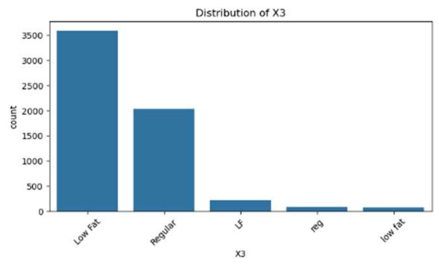


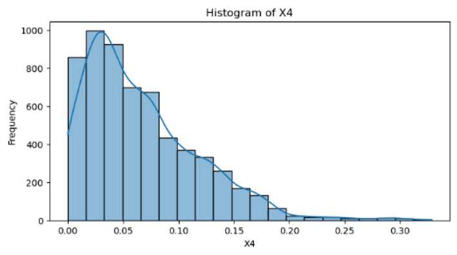
```
Categorical Columns:
Index(['X1', 'X3', 'X5', 'X7', 'X9', 'X10', 'X11'], dtype='object')

Summary of Categorical Data:
           X1       X3                     X5      X7      X9     X10  \
count    6000     6000                   6000    6000    4289    6000
unique   1553        5                     16      10       3       3
top     FDP28  Low Fat  Fruits and Vegetables  OUT045  Medium  Tier 3
freq        8     3595                    875     677    1935    2358
...
count                6000
unique                  4
top     Supermarket Type1
freq                 3967
```

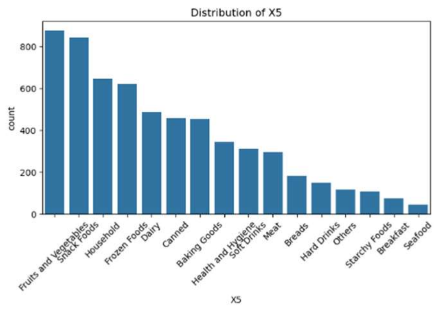
- **Plot Data**

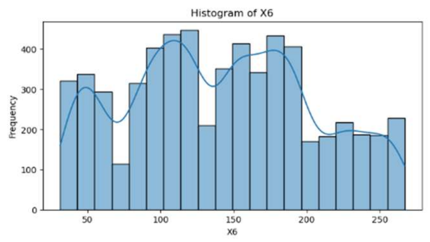

Distribution of X1
count
X1

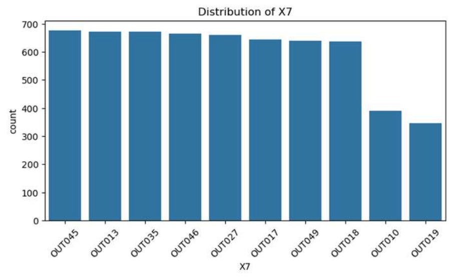

Histogram of X2
Frequency
X2

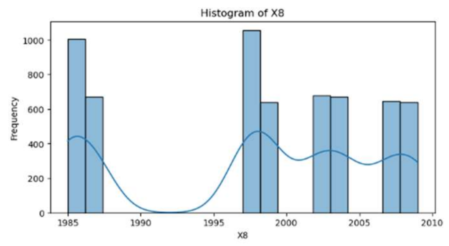

Distribution of X3
count
Low Fat
Regular
LF
reg
low fat
X3

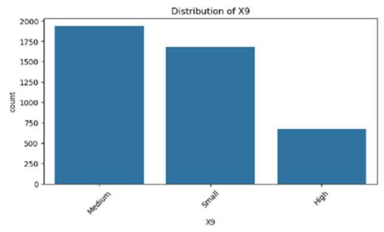

Histogram of X4
Frequency
X4

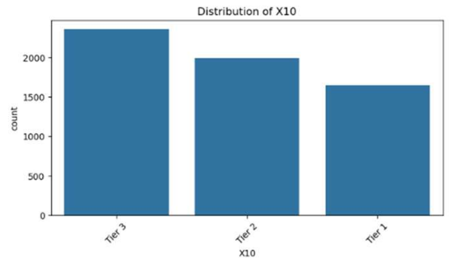

Distribution of X5
count
Fruits and Vegetables
Snack Foods
Household
Frozen Foods
Dairy
Canned
Baking Goods
Health and Hygiene
Soft Drinks
Meat
Breads
Hard Drinks
Others
Starchy Foods
Breakfast
Seafood
X5

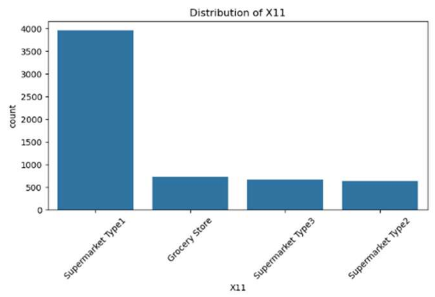

Histogram of X6
Frequency
X6

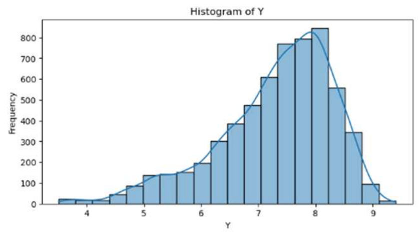

Distribution of X7
count
OUT045
OUT013
OUT035
OUT046
OUT027
OUT017
OUT049
OUT018
OUT010
OUT019
X7


Histogram of X8
Frequency
X8


Distribution of X9
count
Medium
Small
High
X9

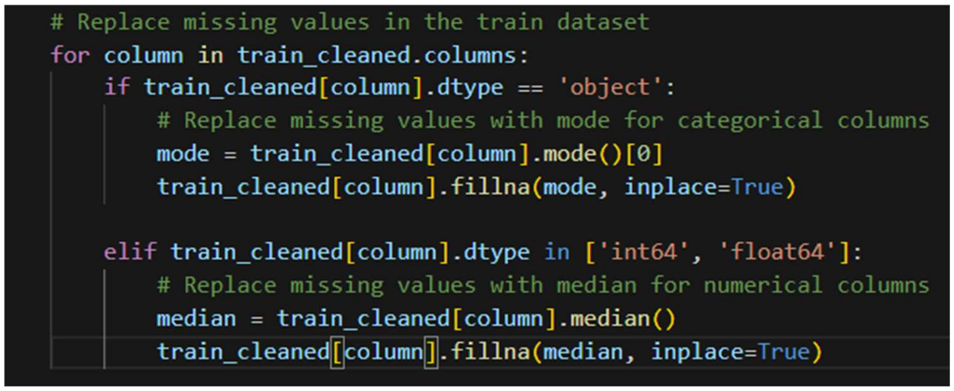

Distribution of X10
count
Tier 3
Tier 2
Tier 1
X10


Distribution of X11
count
Supermarket Type1
Grocery Store
Supermarket Type3
Supermarket Type2
X11


Histogram of Y
Frequency
Y

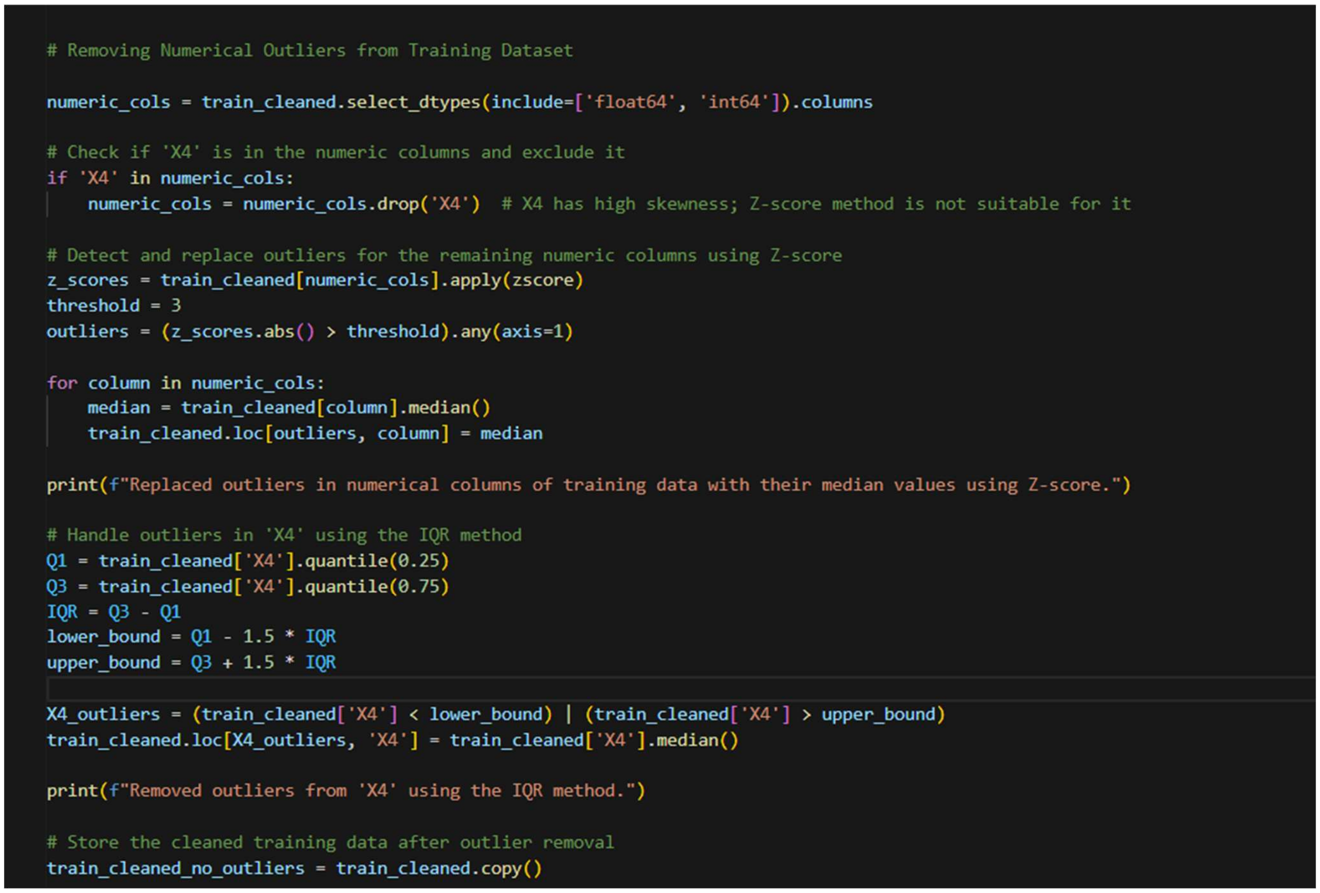
2. DATA PREPROESSING

- **Fill missing values (mean for numeric columns, mode for categorical columns)**



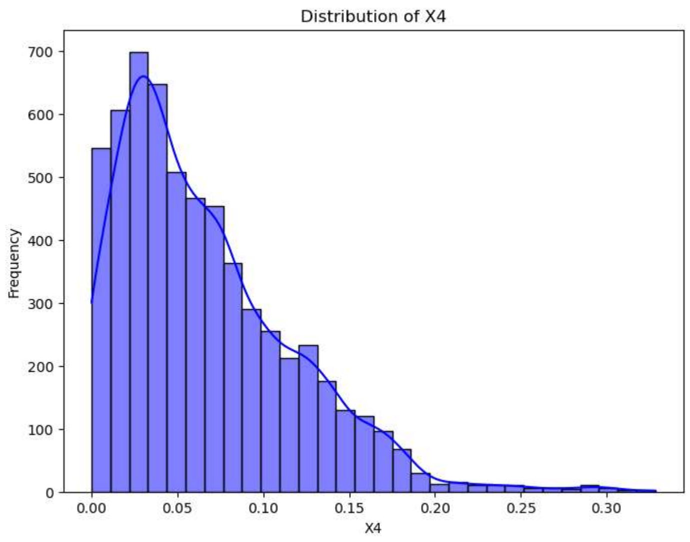
```python
# Replace missing values in the train dataset
for column in train_cleaned.columns:
    if train_cleaned[column].dtype == 'object':
        # Replace missing values with mode for categorical columns
        mode = train_cleaned[column].mode()[0]
        train_cleaned[column].fillna(mode, inplace=True)

    elif train_cleaned[column].dtype in ['int64', 'float64']:
        # Replace missing values with median for numerical columns
        median = train_cleaned[column].median()
        train_cleaned[column].fillna(median, inplace=True)
```

- **Replacing Outliers with Median**



```python
# Removing Numerical Outliers from Training Dataset

numeric_cols = train_cleaned.select_dtypes(include=['float64', 'int64']).columns

# Check if 'X4' is in the numeric columns and exclude it
if 'X4' in numeric_cols:
    numeric_cols = numeric_cols.drop('X4')  # X4 has high skewness; Z-score method is not suitable for it

# Detect and replace outliers for the remaining numeric columns using Z-score
z_scores = train_cleaned[numeric_cols].apply(zscore)
threshold = 3
outliers = (z_scores.abs() > threshold).any(axis=1)

for column in numeric_cols:
    median = train_cleaned[column].median()
    train_cleaned.loc[outliers, column] = median

print(f"Replaced outliers in numerical columns of training data with their median values using Z-score.")

# Handle outliers in 'X4' using the IQR method
Q1 = train_cleaned['X4'].quantile(0.25)
Q3 = train_cleaned['X4'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

X4_outliers = (train_cleaned['X4'] < lower_bound) | (train_cleaned['X4'] > upper_bound)
train_cleaned.loc[X4_outliers, 'X4'] = train_cleaned['X4'].median()

print(f"Removed outliers from 'X4' using the IQR method.")

# Store the cleaned training data after outlier removal
train_cleaned_no_outliers = train_cleaned.copy()
```

In the provided code snippet, outlier removal was implemented using two distinct methods based on the nature of the numeric features:



1. Z-Score Method for Outlier Detection and Replacement

- **Purpose**:
  - This method was applied to all numerical columns except X4 (which exhibited high skewness). The Z-Score method standardizes data and identifies extreme values based on their deviation from the mean.
- **Process**:
  - Z-scores measure how far each data point is from the mean in terms of standard deviations. They are calculated by subtracting the mean and dividing by the standard deviation.
  - A threshold of 3 was set, meaning any value deviating more than 3 standard deviations from the mean was considered an outlier.
  - Outliers were replaced with the median of their respective columns. The median was chosen as it is robust to extreme values, preserving the central tendency of the data.
- **Why It Was Used**:
  - The Z-Score method is effective for detecting outliers in normally distributed features, where values far from the mean are likely anomalies.
  - Replacing outliers with the median ensures that the data distribution remains intact without introducing significant bias.

---

2. IQR Method for Handling X4

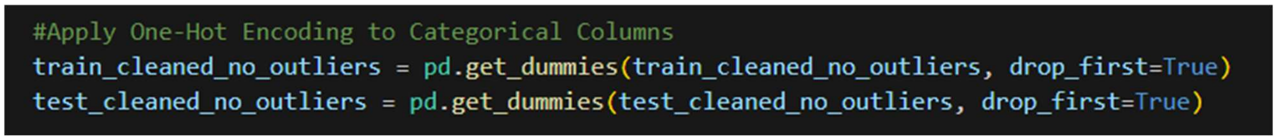
- **Purpose**:
  - The feature X4 was excluded from the Z-Score-based method due to its high skewness. Instead, the Interquartile Range (IQR) method was used to detect and handle outliers.
- **Process**:
  - The Interquartile Range (IQR) was calculated as the difference between the 75th percentile (upper quartile) and the 25th percentile (lower quartile) of the data. Outliers were identified using bounds: the lower bound was set by subtracting 1.5 times the IQR

from the 25th percentile, and the upper bound was set by adding 1.5 times the IQR to the 75th percentile. Data points outside these bounds were considered outliers.Any values below the lower bound or above the upper bound were flagged as outliers.
  - Outliers in X4 were replaced with the column's median to minimize the impact of extreme values.
- **Why It Was Used**:
  - The IQR method is well-suited for features with skewed distributions, as it focuses on the central 50% of data and excludes extreme values without making assumptions about the underlying distribution.
  - Replacing outliers with the median aligns with the need for a robust central measure.

Advantages of This Approach

1. **Custom Handling for Skewed Features**:
   - By using the Z-Score method for normally distributed features and the IQR method for skewed features (X4), the preprocessing is tailored to the data characteristics.
2. **Preservation of Data Integrity**:
   - Replacing outliers with the median ensures that the feature distributions remain close to their original form while mitigating the influence of extreme values.
3. **Robustness**:
   - Both methods are widely used for handling outliers and reduce the risk of biases introduced by extreme values.

• One Hot Encoding



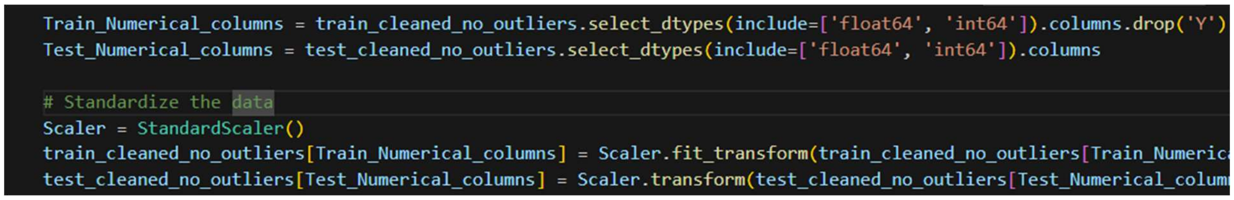
```
#Apply One-Hot Encoding to Categorical Columns
train_cleaned_no_outliers = pd.get_dummies(train_cleaned_no_outliers, drop_first=True)
test_cleaned_no_outliers = pd.get_dummies(test_cleaned_no_outliers, drop_first=True)
```

In this dataset, the categorical columns include: `X1`, `X3`, `X5`, `X7`, `X9`, `X10`, and `X11`. Here's why **one-hot encoding** was chosen and why it matters specifically for this dataset:

Nature of Categorical Variables

- Many of the categorical columns in the dataset (X3, X5, X10, X11) represent nominal data, meaning there is no inherent order or ranking among the categories (e.g., X3 has values like "Low Fat" and "Regular").
  Why It Matters:
- One-hot encoding ensures that each category is represented as an independent binary feature. This prevents the model from interpreting these categories as having an ordinal relationship, which could lead to incorrect assumptions.

Z-Score Normalization (Standardization)



```
Train_Numerical_columns = train_cleaned_no_outliers.select_dtypes(include=['float64', 'int64']).columns.drop('Y')
Test_Numerical_columns = test_cleaned_no_outliers.select_dtypes(include=['float64', 'int64']).columns

# Standardize the data
Scaler = StandardScaler()
train_cleaned_no_outliers[Train_Numerical_columns] = Scaler.fit_transform(train_cleaned_no_outliers[Train_Numerica
test_cleaned_no_outliers[Test_Numerical_columns] = Scaler.transform(test_cleaned_no_outliers[Test_Numerical_colum
```

Standardization, performed using the StandardScaler in this dataset, was a crucial step in preprocessing. This method transforms the numerical data to have a mean of 0 and a standard deviation of 1. Here's why standardization was necessary and beneficial for this dataset:

1. Ensuring Equal Contribution of Features

- Reason:
  - Numerical features in the dataset, such as X2, X4, and X6, likely have varying ranges and magnitudes. For example, X4 could have small values close to zero, while X6 might have larger values in the hundreds or thousands.
  - Without scaling, features with larger values dominate the optimization process, making smaller-scale features less influential.
- Why It Matters for This Dataset:
  - By standardizing all numerical columns, each feature contributes equally to the model's performance.

2. Suitability for Distance-Based Models

- **Reason**:
  - Models like SVM rely on distance calculations (e.g., Euclidean distance) to separate data points. If the features are not standardized, one feature's larger range could skew the distance metric, resulting in inaccurate model boundaries.
- **Why It Matters for This Dataset**:
  - Features like X4 with small values and X6 with larger values could disrupt the SVM kernel's computation. StandardScaler ensures that all features have an equal impact on the decision boundary.

3. FEATURE SELECTION

- **Feature selection was performed using a correlation analysis to identify the most relevant features for predicting the target variable (Y). Here's a detailed explanation of the process:**

1. Correlation Calculation

- What Was Done:
  - The correlation between each feature and the target variable (Y) was computed using the .corr() method.
  - This method calculates Pearson correlation coefficients, which measure the linear relationship between two variables. Values range from:
    - +1: Perfect positive correlation (as one increases, the other increases).
    - -1: Perfect negative correlation (as one increases, the other decreases).
    - 0: No linear relationship.
- Why This Matters:
  - Features with high absolute correlation values (closer to +1 or -1) are more predictive of the target variable and are prioritized for inclusion in the model.

2. Sorting Features by Correlation

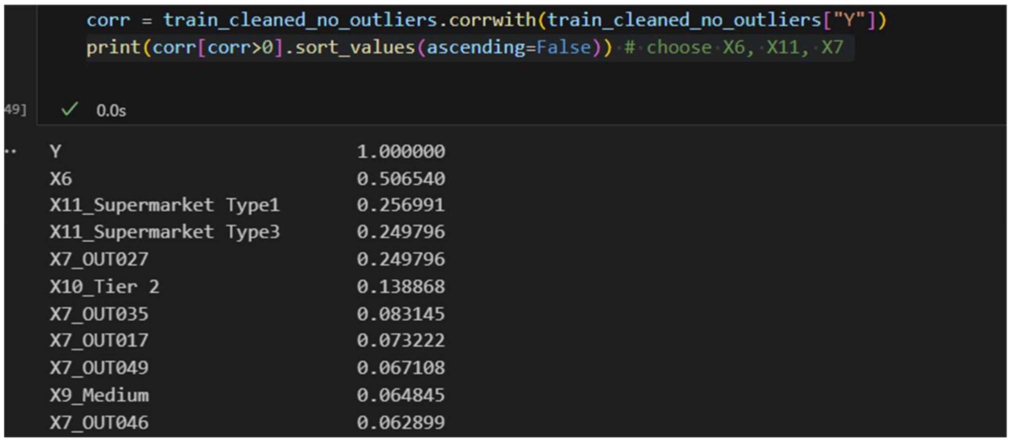
What Was Done:

Correlation values were sorted in descending order to identify features with the strongest positive correlation with Y.

A subset of features with correlation values greater than a threshold (e.g., > 0) was selected for further analysis.

Why This Matters:

Sorting helps quickly identify the most important predictors and exclude weakly correlated or irrelevant features.



```
corr = train_cleaned_no_outliers.corrwith(train_cleaned_no_outliers["Y"])
print(corr[corr>0].sort_values(ascending=False)) # choose X6, X11, X7
```

```
[49] ✓ 0.0s
```

```
Y                        1.000000
X6                       0.506540
X11_Supermarket Type1    0.256991
X11_Supermarket Type3    0.249796
X7_OUT027                0.249796
X10_Tier 2               0.138868
X7_OUT035                0.083145
X7_OUT017                0.073222
X7_OUT049                0.067108
X9_Medium                0.064845
X7_OUT046                0.062899
```

3. Removing Weakly Correlated Features

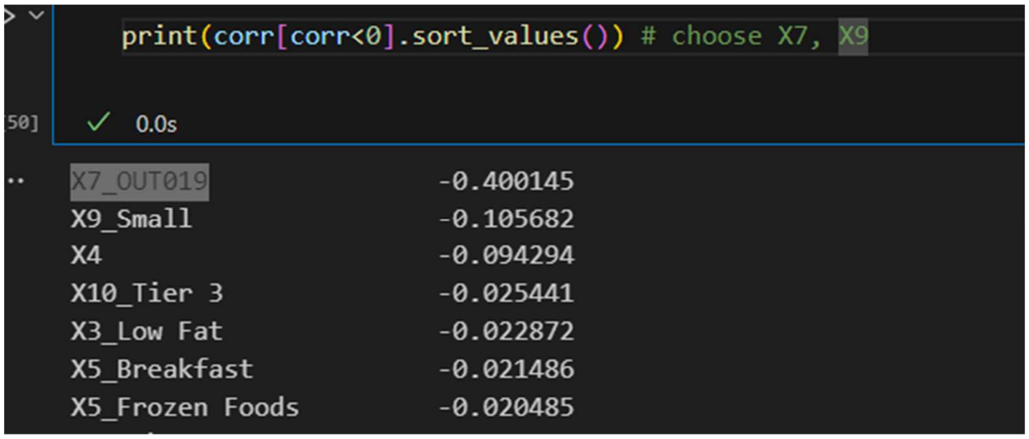
What Was Done:

Features with low or negative correlation values (e.g., X7_OUT019, X9_Small, X4) were excluded as they contribute little or no useful information to the model.

These features were dropped from the dataset to simplify the model and reduce noise.

Why This Matters:

Including irrelevant features can increase model complexity, reduce performance, and lead to overfitting. Removing these features improves efficiency and accuracy.



```
print(corr[corr<0].sort_values()) # choose X7, X9
```

```
[50] ✓ 0.0s
```

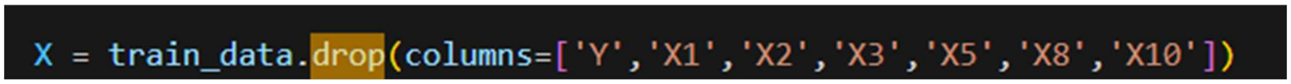
```
X7_OUT019             -0.400145
X9_Small              -0.105682
X4                    -0.094294
X10_Tier 3            -0.025441
X3_Low Fat            -0.022872
X5_Breakfast          -0.021486
X5_Frozen Foods       -0.020485
```

5. Final Feature Selection

The final set of features for the model was determined by dropping irrelevant columns, including features with weak or no correlation with Y:

Dropped columns: ['Y', 'X1', 'X2', 'X3', 'X5', 'X8', 'X10'].

Retained features include strong predictors such as X6, X7, and X11.



```
X = train_data.drop(columns=['Y','X1','X2','X3','X5','X8','X10'])
```

4.Hyperparameter Tuning

In this project, hyperparameter tuning was performed using GridSearchCV for some models and Optuna for others. Each method was chosen based on the complexity of the model and the size of the hyperparameter space.

1. Models Using GridSearchCV

- **Process**:
  - A predefined grid of hyperparameters was created, and GridSearchCV tested all combinations using k-fold cross-validation (e.g., 5-fold).
  - This method systematically searched the hyperparameter space to identify the best-performing combination.
- **When It Was Used**:
  - For models with smaller or well-defined hyperparameter spaces, such as:
    - **Support Vector Regression (SVR)**: Parameters like C, gamma, epsilon, and kernel type were tuned.
    - **Random Forest**: Parameters such as the number of estimators, maximum depth, and minimum samples split were optimized.
- **Advantages**:
  - Exhaustive and systematic search ensures thorough exploration.
  - Well-suited for models where the hyperparameter space is discrete and manageable in size.

2. Models Using Optuna

- **Process**:
  - Optuna, a framework for Bayesian optimization, dynamically explored the hyperparameter space by learning from previous iterations.
  - It prioritized promising hyperparameter ranges, reducing computation time compared to exhaustive methods.
- **When It Was Used**:

  - For models with larger or continuous hyperparameter spaces, such as tree-based ensemble methods (e.g., Gradient Boosting).
  - Parameters like learning rate, maximum depth, and the number of estimators were tuned using Optuna.
- **Advantages**:
  - Efficient for large or continuous parameter spaces.
  - Quickly identifies optimal configurations with fewer evaluations, saving computational resources.

Evaluation Metrics

- Both GridSearchCV and Optuna used **Mean Absolute Error (MAE)** as the evaluation metric to ensure consistency across all models. MAE was chosen for its interpretability and robustness to outliers.

Why Two Tuning Methods Were Used

- **GridSearchCV**: Ideal for simpler models or when the parameter space is discrete and small, such as Random Forest and SVR.
- **Optuna**: Better for more complex models or when the hyperparameter space is large and continuous, such as Gradient Boosting.

  By using the most appropriate tuning method for each model, the project achieved optimal performance while balancing computational efficiency.

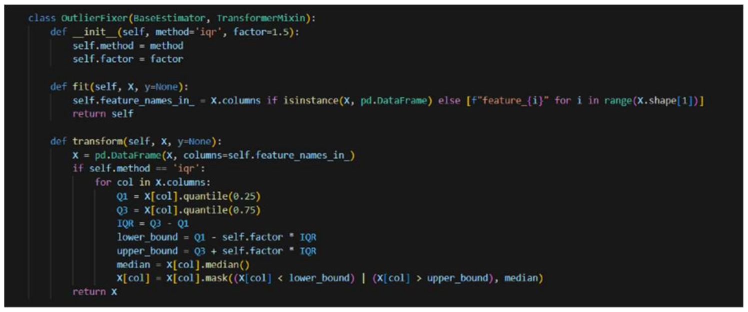
5.PREPROCESSING USING PIPELINING

Preprocessing in this project in some models was made efficient and consistent by implementing **pipelines**. These pipelines integrated multiple preprocessing steps for numerical and categorical features, ensuring that all transformations were applied systematically during both training and testing. Below is a detailed explanation of the components and their roles:

1. Outlier Handling with a Custom Transformer

Custom Transformer: OutlierFixer

- **Purpose**:
  - The OutlierFixer class is a custom transformer used to identify and handle outliers in numerical features using the **Interquartile Range (IQR)** method.
- **How It Works**:
  - For each feature, the IQR is calculated as the difference between the 75th percentile (Q3) and the 25th percentile (Q1).
  - Outlier bounds are computed:
    - Lower Bound = Q1 - (factor × IQR)
    - Upper Bound = Q3 + (factor × IQR)
  - Values outside these bounds are replaced with the median of the feature.



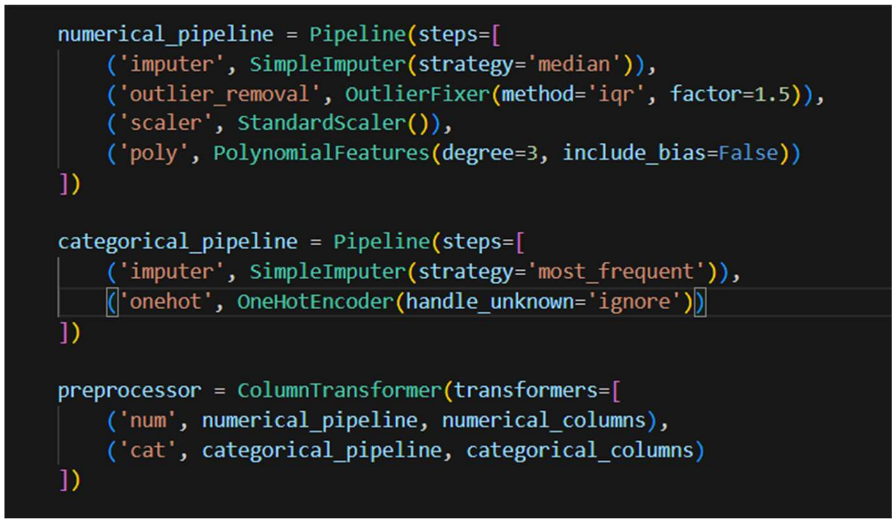
```python
class OutlierFixer(BaseEstimator, TransformerMixin):
    def __init__(self, method='iqr', factor=1.5):
        self.method = method
        self.factor = factor

    def fit(self, X, y=None):
        self.feature_names_in_ = X.columns if isinstance(X, pd.DataFrame) else [f"feature_{i}" for i in range(X.shape[1])]
        return self

    def transform(self, X, y=None):
        X = pd.DataFrame(X, columns=self.feature_names_in_)
        if self.method == 'iqr':
            for col in X.columns:
                Q1 = X[col].quantile(0.25)
                Q3 = X[col].quantile(0.75)
                IQR = Q3 - Q1
                lower_bound = Q1 - self.factor * IQR
                upper_bound = Q3 + self.factor * IQR
                median = X[col].median()
                X[col] = X[col].mask((X[col] < lower_bound) | (X[col] > upper_bound), median)
        return X
```

- **Advantages**:
  - By encapsulating the outlier handling logic into a transformer, it can be seamlessly integrated into a pipeline and reused across different datasets.
  - Ensures consistency in how outliers are treated, improving the robustness of the model.

2. Numerical Feature Pipeline

Steps in the Numerical Pipeline:

1. **Imputation**:
   - **Method**: SimpleImputer with strategy='median'.
   - **Purpose**: Fills missing values in numerical features with their median, which is robust to outliers.
2. **Outlier Removal**:
   - **Method**: OutlierFixer (custom transformer).
   - **Purpose**: Replaces extreme values with the median, as described above.



```python
numerical_pipeline = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('outlier_removal', OutlierFixer(method='iqr', factor=1.5)),
    ('scaler', StandardScaler()),
    ('poly', PolynomialFeatures(degree=3, include_bias=False))
])

categorical_pipeline = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])

preprocessor = ColumnTransformer(transformers=[
    ('num', numerical_pipeline, numerical_columns),
    ('cat', categorical_pipeline, categorical_columns)
])
```

3. **Scaling**:
    - **Method**: StandardScaler.
    - **Purpose**: Normalizes numerical features to have zero mean and unit variance, ensuring equal contribution during model training.
4. **Polynomial Feature Generation**:
    - **Method**: PolynomialFeatures with degree=3.
    - **Purpose**: Generates interaction terms and non-linear features to capture complex relationships.

3. Categorical Feature Pipeline

Steps in the Categorical Pipeline:

1. **Imputation**:
    - **Method**: SimpleImputer with strategy='most_frequent'.
    - **Purpose**: Fills missing values in categorical features with the most frequent category, preserving the mode of the distribution.
2. **One-Hot Encoding**:
    - **Method**: OneHotEncoder with handle_unknown='ignore'.
    - **Purpose**: Converts categorical features into binary columns, ensuring numerical compatibility with machine learning algorithms. It ignores unseen categories in the test set, avoiding errors.

Combining the Pipelines with a ColumnTransformer

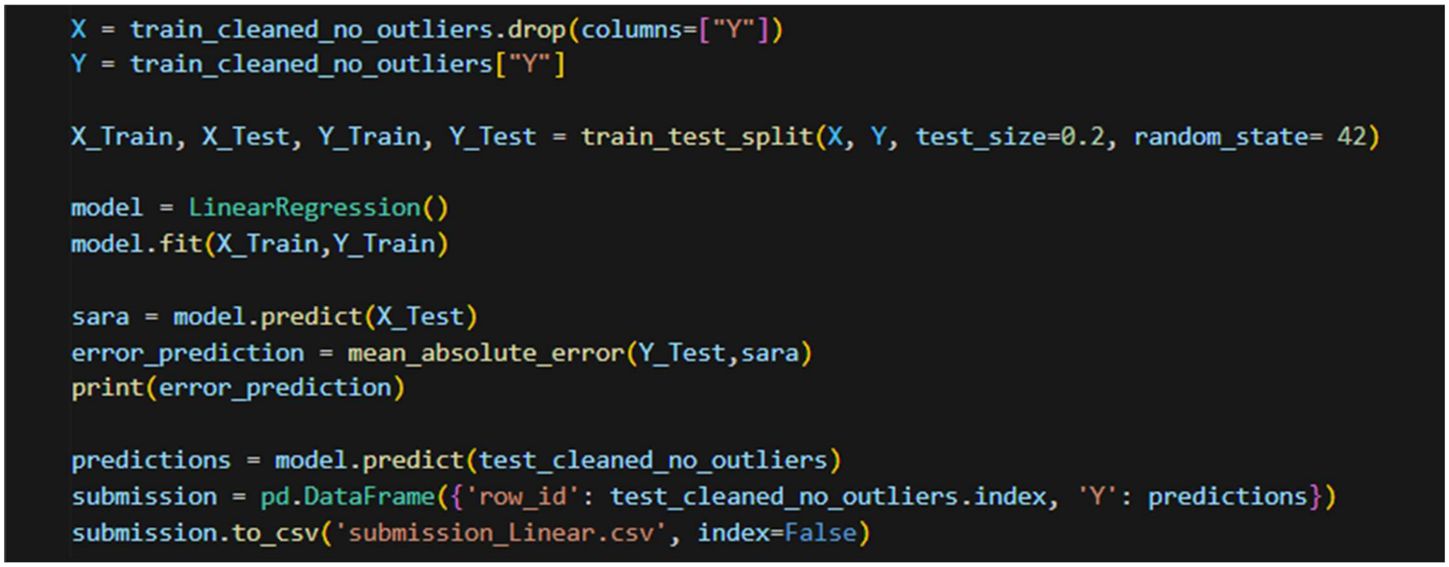
- **Implementation**:
    - A ColumnTransformer was used to combine the numerical and categorical pipelines.
    - It applied the numerical pipeline to specified numerical columns and the categorical pipeline to categorical columns, ensuring appropriate preprocessing for each data type.
- **Benefits**:
    - Simplifies preprocessing by managing transformations for different data types within a single structure.
    - Ensures that numerical and categorical features are processed separately but consistently.

The preprocessing pipeline was specifically designed for **Random Forest** and **XGBoost** to address the complexity of their training requirements. By automating and integrating preprocessing steps, the pipeline ensured consistent, clean data for training, enhancing the models' ability to capture meaningful patterns and deliver robust predictions.

6.0 MODELS

6.1 Linear Regression

Linear Regression was chosen as one of the baseline models for this dataset to predict the target variable (Y). The following steps outline the training process and its key components:



```
X = train_cleaned_no_outliers.drop(columns=["Y"])
Y = train_cleaned_no_outliers["Y"]

X_Train, X_Test, Y_Train, Y_Test = train_test_split(X, Y, test_size=0.2, random_state= 42)

model = LinearRegression()
model.fit(X_Train,Y_Train)

sara = model.predict(X_Test)
error_prediction = mean_absolute_error(Y_Test,sara)
print(error_prediction)

predictions = model.predict(test_cleaned_no_outliers)
submission = pd.DataFrame({'row_id': test_cleaned_no_outliers.index, 'Y': predictions})
submission.to_csv('submission_Linear.csv', index=False)
```

1- Data Splitting:

The dataset was divided into independent variables (X) and the target variable (Y).

train_test_split split the data into 80% training and 20% testing subsets to ensure robust evaluation.

2- Model Training:

A LinearRegression model was instantiated and fit to the training data (X_Train and Y_Train) using the .fit() method.

This step calculated the best-fit coefficients for minimizing prediction errors.

3- Model Evaluation:

Predictions were made on the test set (X_Test) and evaluated using Mean Absolute Error (MAE).

MAE was chosen because it is simple, interpretable, and robust to outliers, providing the average magnitude of prediction errors in the same units as the target variable.

4- Prediction and Submission:

Predictions were generated on cleaned test data, and the results were saved in a CSV file for submission.

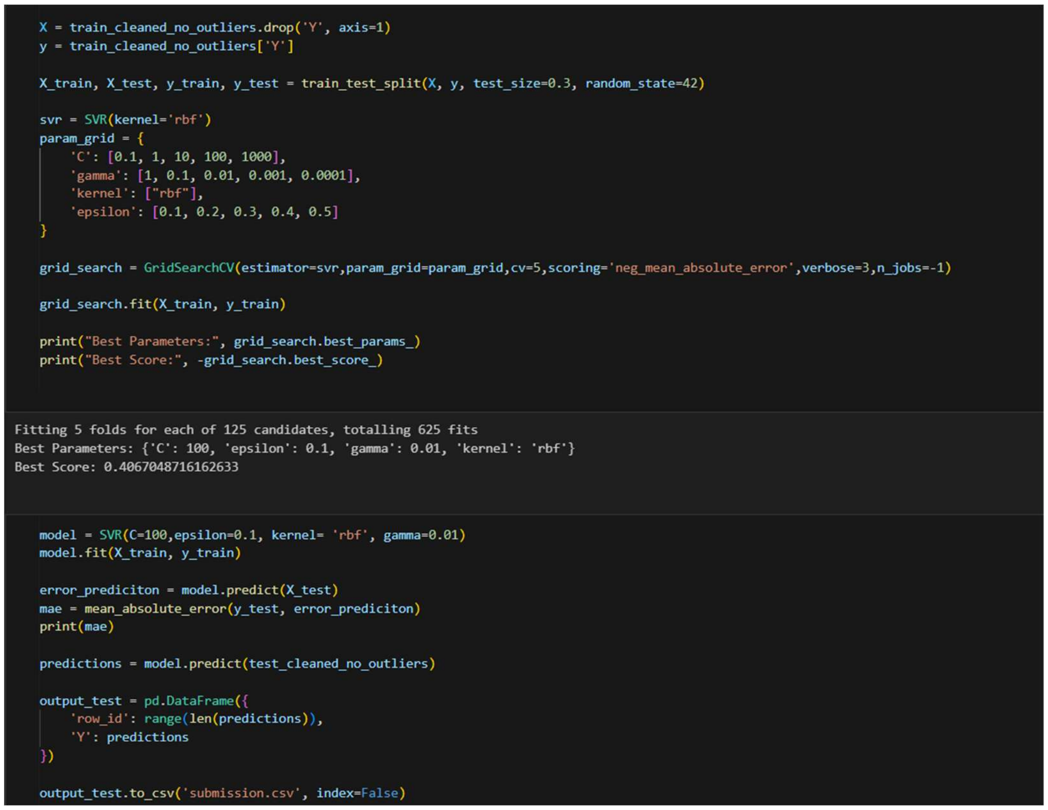
6.2 Support Vector Machine SVR

Support Vector Regression (SVR) was used in this project to model the relationship between the input features and the target variable. SVR is a powerful regression algorithm that extends Support Vector Machines (SVM) to continuous target prediction. It is particularly effective for handling non-linear relationships in data, thanks to its use of kernel functions.

Key Features of SVR

1. **Kernel Functions**:
   - The **Radial Basis Function (RBF)** kernel was chosen for this task, as it is well-suited for capturing non-linear patterns in the data.
   - The RBF kernel maps the input features into a higher-dimensional space where linear regression can effectively capture complex relationships.
2. **Margin of Tolerance (epsilon)**:
   - SVR introduces an epsilon margin, within which predictions are considered acceptable without penalty. This margin ensures the model focuses on larger deviations, improving robustness.
3. **Regularization Parameter (C)**:
   - Controls the trade-off between achieving a low error on the training set and maintaining a simple model that generalizes well to new data.
   - A higher C reduces bias by focusing more on minimizing training errors but risks overfitting.
4. **Gamma**:
   - Defines the influence of individual data points in the RBF kernel. Lower values result in a smoother model, while higher values make the model more sensitive to specific data points.

Implementation:



```
X = train_cleaned_no_outliers.drop('Y', axis=1)
y = train_cleaned_no_outliers['Y']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

svr = SVR(kernel='rbf')
param_grid = {
    'C': [0.1, 1, 10, 100, 1000],
    'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
    'kernel': ["rbf"],
    'epsilon': [0.1, 0.2, 0.3, 0.4, 0.5]
}

grid_search = GridSearchCV(estimator=svr,param_grid=param_grid,cv=5,scoring='neg_mean_absolute_error',verbose=3,n_jobs=-1)

grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", -grid_search.best_score_)
```

```
Fitting 5 folds for each of 125 candidates, totalling 625 fits
Best Parameters: {'C': 100, 'epsilon': 0.1, 'gamma': 0.01, 'kernel': 'rbf'}
Best Score: 0.4067048716162633
```

```
model = SVR(C=100,epsilon=0.1, kernel= 'rbf', gamma=0.01)
model.fit(X_train, y_train)

error_prediciton = model.predict(X_test)
mae = mean_absolute_error(y_test, error_prediciton)
print(mae)

predictions = model.predict(test_cleaned_no_outliers)

output_test = pd.DataFrame({
    'row_id': range(len(predictions)),
    'Y': predictions
})

output_test.to_csv('submission.csv', index=False)
```

- **Grid Search Execution**:
  - GridSearchCV performed an exhaustive search across all combinations of the hyperparameters using 5-fold cross-validation.
  - The performance was evaluated using negative Mean Absolute Error (neg_mean_absolute_error) to ensure consistency.
- **Best Parameters**:
  - The best parameter combination identified was:
    - C: 100
    - gamma: 0.01
    - epsilon: 0.1
    - kernel: 'rbf'
  - The corresponding cross-validation score was approximately 0.407.

3. Model Training

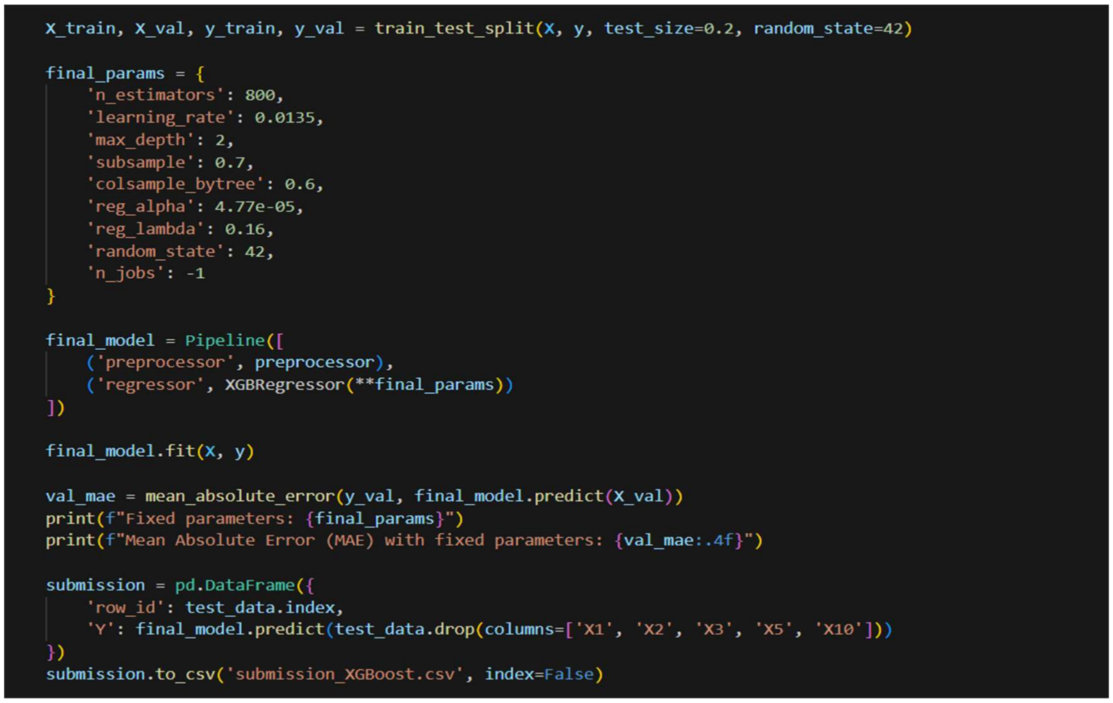
- **Final Model Configuration**:
  - The SVR model was re-instantiated with the best parameters found during grid search.
  - The model was trained on the training data (X_train and y_train) using the .fit() method.

4. Model Evaluation

- **Test Set Predictions**:
  - The model predicted the target values for the test set (X_test) using the .predict() method.
- **Evaluation Metric**:
  - The **Mean Absolute Error (MAE)** was calculated between the actual (y_test) and predicted values, measuring the average magnitude of prediction errors.
  - This metric was chosen for its simplicity and interpretability.

6.3 Gradient Boosting using XGBoost

XGBoost (Extreme Gradient Boosting) is a powerful and efficient machine learning algorithm designed for both regression and classification tasks. It is based on the concept of **gradient boosting**, where models are built sequentially to minimize errors from previous iterations. Each new tree corrects the errors made by the ensemble of existing trees.



```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

final_params = {
    'n_estimators': 800,
    'learning_rate': 0.0135,
    'max_depth': 2,
    'subsample': 0.7,
    'colsample_bytree': 0.6,
    'reg_alpha': 4.77e-05,
    'reg_lambda': 0.16,
    'random_state': 42,
    'n_jobs': -1
}

final_model = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(**final_params))
])

final_model.fit(X, y)

val_mae = mean_absolute_error(y_val, final_model.predict(X_val))
print(f"Fixed parameters: {final_params}")
print(f"Mean Absolute Error (MAE) with fixed parameters: {val_mae:.4f}")

submission = pd.DataFrame({
    'row_id': test_data.index,
    'Y': final_model.predict(test_data.drop(columns=['X1', 'X2', 'X3', 'X5', 'X10']))
})
submission.to_csv('submission_XGBoost.csv', index=False)
```

Key Features:

1. **Gradient Boosting Framework**:
    - Builds decision trees iteratively, each tree reducing the residual errors of the previous trees.
2. **Regularization**:
    - Includes reg_alpha (L1 regularization) and reg_lambda (L2 regularization) to prevent overfitting.
3. **Parallel Computing**:
    - Optimized for speed with parallel processing and efficient memory usage.
4. **Customizable Hyperparameters**:
    - Parameters like learning_rate, n_estimators, max_depth, and others allow fine-tuning for performance.
5. **Handling Missing Data**:
    - Automatically handles missing values during training.

Implementation:

1. **Data Splitting**:
    - The dataset was split into training (X_train, y_train) and validation (X_val, y_val) sets using an 80-20 ratio. This ensured that model performance was evaluated on unseen data.
2. **Hyperparameter Configuration**:
    - A predefined set of optimized hyperparameters via Optuna (final_params) was used to initialize the XGBoost model. These included:
        - n_estimators: Number of trees in the ensemble (800).
        - learning_rate: Step size for updating weights (0.0135).
        - max_depth: Maximum depth of a tree (2).
        - subsample: Fraction of samples used for training each tree (0.7).
        - colsample_bytree: Fraction of features used for each tree (0.6).
        - reg_alpha and reg_lambda: Regularization terms to reduce overfitting.
3. **Pipeline Integration**:
    - A Pipeline was created to combine preprocessing steps with the XGBoost model:
        - **Preprocessor**: Handles feature transformations (e.g., scaling, encoding).

        - **Regressor**: The XGBoost model configured with the hyperparameters.
4. **Model Training**:
    - The final_model pipeline was fit to the training data (X_train and y_train) using the .fit() method.
5. **Evaluation**:
    - Predictions were made on the validation set (X_val) using the .predict() method.
    - The **Mean Absolute Error (MAE)** was computed to evaluate the model's performance. MAE measures the average magnitude of prediction errors, offering an interpretable metric.
6. **Submission Preparation**:
    - Predictions on the test dataset (test_data) were generated using the trained pipeline.
    - A DataFrame containing row indices and predicted values was saved as a CSV file (submission_XGBoost.csv) for submission.

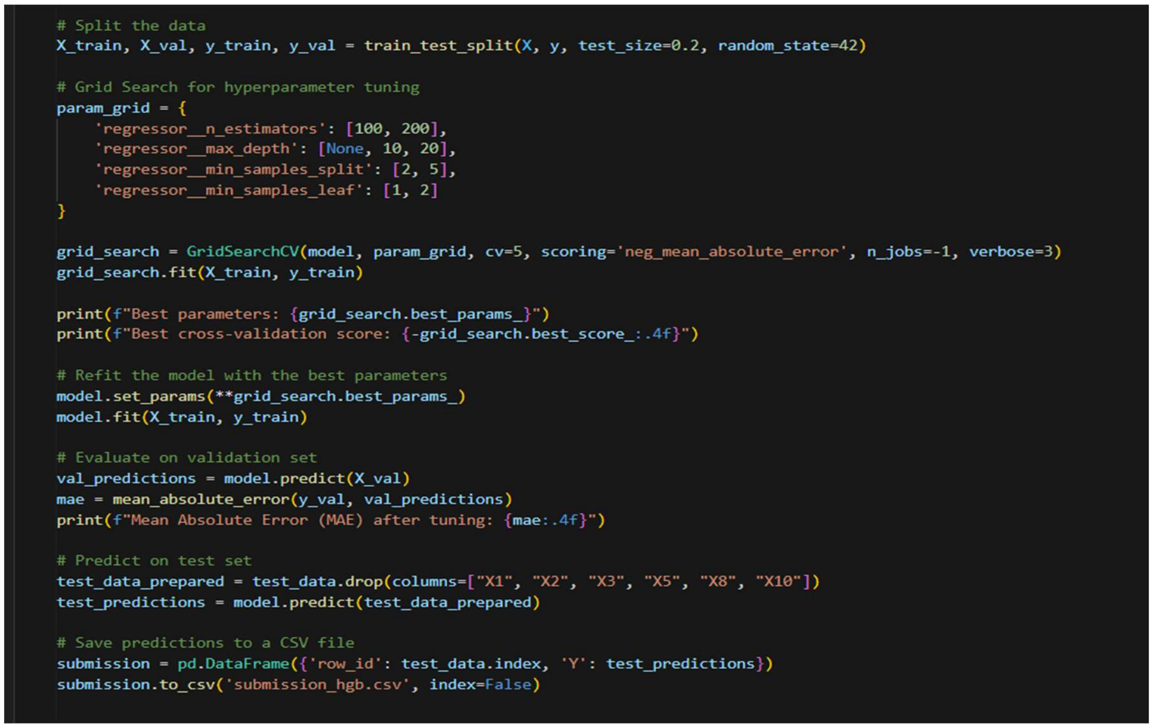
Advantages of XGBoost

- Handles non-linear relationships effectively.
- Prevents overfitting through regularization.
- Scales well to large datasets due to its efficient implementation.
- Supports advanced features like feature importance and missing value handling

6.4 Random Forrest

The Random Forest model was implemented using a similar preprocessing pipeline as XGBoost. Random Forest is an ensemble learning method that builds multiple decision trees and aggregates their outputs to improve prediction accuracy and reduce overfitting. Here's a detailed breakdown of the Random Forest implementation:

Training and Hyperparameter Tuning:



```
# Split the data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Grid Search for hyperparameter tuning
param_grid = {
    'regressor__n_estimators': [100, 200],
    'regressor__max_depth': [None, 10, 20],
    'regressor__min_samples_split': [2, 5],
    'regressor__min_samples_leaf': [1, 2]
}

grid_search = GridSearchCV(model, param_grid, cv=5, scoring='neg_mean_absolute_error', n_jobs=-1, verbose=3)
grid_search.fit(X_train, y_train)

print(f"Best parameters: {grid_search.best_params_}")
print(f"Best cross-validation score: {-grid_search.best_score_:.4f}")

# Refit the model with the best parameters
model.set_params(**grid_search.best_params_)
model.fit(X_train, y_train)

# Evaluate on validation set
val_predictions = model.predict(X_val)
mae = mean_absolute_error(y_val, val_predictions)
print(f"Mean Absolute Error (MAE) after tuning: {mae:.4f}")

# Predict on test set
test_data_prepared = test_data.drop(columns=["X1", "X2", "X3", "X5", "X8", "X10"])
test_predictions = model.predict(test_data_prepared)

# Save predictions to a CSV file
submission = pd.DataFrame({'row_id': test_data.index, 'Y': test_predictions})
submission.to_csv('submission_hgb.csv', index=False)
```

Hyperparameter Tuning with GridSearchCV:

- A grid of hyperparameters was defined to optimize the model:
  - n_estimators: Number of trees in the forest (e.g., 100 to 200).
  - max_depth: Maximum depth of the trees to control overfitting.
  - min_samples_split: Minimum number of samples required to split a node.
  - min_samples_leaf: Minimum number of samples required to be a leaf node.
- **GridSearchCV**:
  - Conducted 5-fold cross-validation to evaluate combinations of these parameters using the negative Mean Absolute Error (neg_mean_absolute_error) as the scoring metric.
  - The best parameters were identified and used to refit the model.

3. Model Training

- After determining the best hyperparameters, the Random Forest model was retrained on the training set (X_train and y_train) using the .fit() method.
- This step ensured the model was optimized for the given dataset.

4. Evaluation

- **Validation Set**:

    - Predictions were made on the validation set (X_val) using the .predict() method.
    - The **Mean Absolute Error (MAE)** was computed to measure the average magnitude of prediction errors.
- **Test Set Predictions**:
    - Predictions were generated on the cleaned test data (test_data_prepared) after applying the preprocessing pipeline.

5. Submission

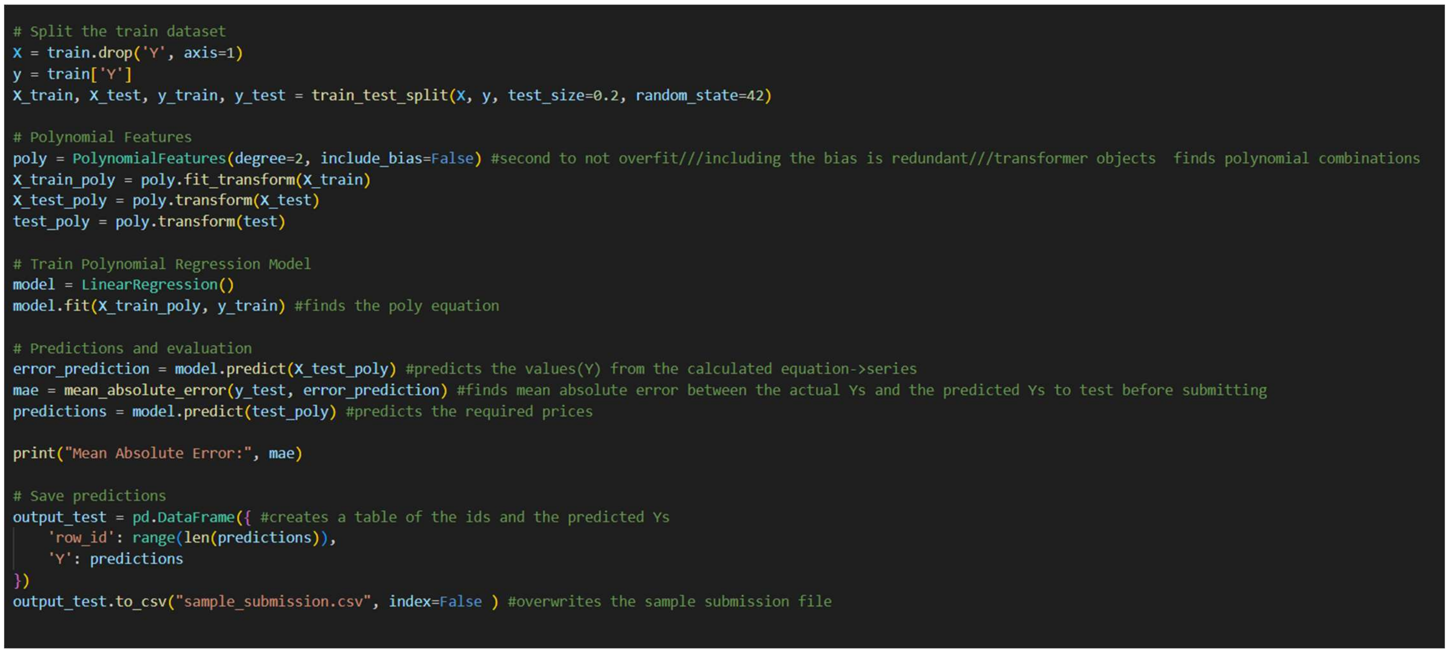
- The test predictions were saved in a CSV file for submission, with the format:
    - row_id: Row indices of the test data.
    - Y: Predicted target values.

Advantages of Random Forest

1. **Robust to Overfitting**:
    - By aggregating multiple trees, Random Forest reduces variance and prevents overfitting, especially when the dataset is noisy.
2. **Handles Mixed Data Types**:
    - Works effectively with both numerical and categorical features, especially when preprocessed consistently.
3. **Interpretability**:
    - Provides feature importance scores, allowing insights into the most influential predictors.

6.5 Polynomial Regression

Polynomial Regression extends linear regression by incorporating polynomial terms of the features, allowing the model to capture non-linear relationships between the predictors and the target variable. Here's an overview of how Polynomial Regression was implemented in this project:



```
# Split the train dataset
X = train.drop('Y', axis=1)
y = train['Y']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Polynomial Features
poly = PolynomialFeatures(degree=2, include_bias=False) #second to not overfit///including the bias is redundant///transformer objects  finds polynomial combinations
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)
test_poly = poly.transform(test)

# Train Polynomial Regression Model
model = LinearRegression()
model.fit(X_train_poly, y_train) #finds the poly equation

# Predictions and evaluation
error_prediction = model.predict(X_test_poly) #predicts the values(Y) from the calculated equation->series
mae = mean_absolute_error(y_test, error_prediction) #finds mean absolute error between the actual Ys and the predicted Ys to test before submitting
predictions = model.predict(test_poly) #predicts the required prices

print("Mean Absolute Error:", mae)

# Save predictions
output_test = pd.DataFrame({ #creates a table of the ids and the predicted Ys
    'row_id': range(len(predictions)),
    'Y': predictions
})
output_test.to_csv("sample_submission.csv", index=False ) #overwrites the sample submission file
```

1.Preprocessing with Polynomial Features

- **Generating Polynomial Features**:
  - The PolynomialFeatures transformer was used to generate polynomial terms (degree=2) for each feature in the dataset.
  - Interaction terms between features were included, enabling the model to capture non-linear relationships.
  - **Key Parameters**:
    - degree=2: Specifies the degree of the polynomial features.
    - include_bias=False: Excludes the bias term to avoid redundancy, as it is already accounted for in the regression model.
- **Transformations**:
  - fit_transform was applied to the training data to generate polynomial terms.
  - The same transformation was applied to the test and validation datasets using transform to ensure consistency.

2. Model Training

- **Algorithm**:
  - A LinearRegression model was used to fit the polynomial-transformed data (X_train_poly).
  - Polynomial Regression works by solving the linear regression problem in the expanded feature space created by the polynomial terms.
- **Process**:
  - The model learned the coefficients for each polynomial term to minimize the prediction error on the training data.

3. Evaluation

- **Validation**:
  - Predictions were made on the validation set (X_test_poly) using the .predict() method.
  - The **Mean Absolute Error (MAE)** was calculated to evaluate the model's performance. MAE measures the average magnitude of prediction errors, providing an interpretable and robust metric.
- **Test Predictions**:
  - The trained model was used to generate predictions for the test set (test_poly), which were prepared for submission.

4. Submission

- **Output Preparation**:
  - A DataFrame was created containing the row_id and predicted values (Y).
  - The results were saved as a CSV file (sample_submission.csv) for submission.

Advantages of Polynomial Regression

1. **Capturing Non-Linear Relationships**:
   - By introducing polynomial terms, the model can fit data with non-linear patterns, which cannot be captured by standard linear regression.
2. **Simplicity**:
   - Polynomial Regression retains the simplicity of linear regression while expanding its capabilities.
3. **Flexibility**:
   - The degree of the polynomial can be adjusted to balance model complexity and overfitting.

7.0 CONCLUSION

This project successfully built an AI system to predict product prices using different machine learning models, including **Linear Regression**, **Polynomial Regression**, **Support Vector Regression (SVR)**, **XGBoost**, and **Random Forest**. Each model was tailored to handle the dataset's specific challenges, ensuring accurate and reliable predictions.

Key Achievements

1. **Data Preprocessing**:
   - Advanced preprocessing, such as outlier removal, scaling, and encoding, ensured clean and consistent data for all models.
   - Pipelines simplified preprocessing and prevented data issues.
2. **Model Optimization**:
   - GridSearchCV and Optuna were used to fine-tune model parameters, improving their performance.
   - Feature engineering and selection focused on the most important variables.
3. **Performance**:
   - Models like XGBoost and Random Forest performed well, handling complex patterns and non-linear relationships effectively.
   - MAE (Mean Absolute Error) was used to evaluate all models, ensuring consistent and interpretable results.

Next Steps

- Combine the strengths of different models using ensemble techniques.
- Explore new features or additional data to improve predictions further.
- Implement the best-performing model for real-world applications.

This project shows how thoughtful preprocessing, model selection, and tuning can create accurate AI solutions for real-world problems like price prediction.