Ain Shams University – Faculty of Engineering I-CHEP

Machine Learning Model

Report

Introduction to Artificial Intelligence – CSE-281

December 2024

Submitted to:

Dr. Mariam Nabil

Eng. Mohammad Essam

Submitted by:

- 23P0270: Ahmed Sherif Nasri
- 23P0314: Omar Ahmed Shabaan
- 2300535: Mohamed Tarek Elsayed
- 23P0182: Peter Maged Shokry
- 23P0282: Mohamed Ahmed Mohamed

23P0171: Omar Montaser Mahmoud

---

1 Contents

2 Data Analysis

2.1 Overview

Dataset Size: The dataset contains 6000 entries with multiple numerical and categorical features.

Target Variable: Y (specifics not provided, assumed to be the dependent variable for prediction).

2.2 Key Insights

Feature X1 (Item Identifier):

- Contains 1553 unique identifiers.
- High cardinality necessitated feature splitting into `Category`, `Subcategory`, and `SKU`.
- Categories reduced to manageable values (e.g., 3 for `Category`).

Feature X2 (Numerical):

- Missing values (1006) handled by imputing the mean.
- Statistical Summary: Mean = 12.96, Std Dev = 4.65, Min = 4.55, Max = 21.35.

Feature X3 (Fat Content):

- Inconsistent labels cleaned and unified into `LF` (Low Fat) and `REG` (Regular).
- Distribution: 3889 (`LF`), 2111 (`REG`).

Feature X4 (Visibility):

- Positively skewed data with many zeros (360 occurrences).
- Log transformation considered for normalization.

Feature X7 (Outlet):

- Even distribution across 10 unique outlets.
- Encoding strategies tested: ordinal encoding and one-hot encoding.

Feature X8 (Establishment Year):

- Transformed to `PastYearsCont` (Years since 2009) and binned into categorical groups (`very old`, `old`, `recent`, `new`).

Other Key Features:

- X5: Product category (16 unique values).
- X9: Size (`Small`, `Medium`, `High`) transformed using ordinal encoding.
- X10: Location type (`Tier 1`, `Tier 2`, `Tier 3`), assumed to reflect luxury.

3 Feature Selection

3.1 Initial Selection

Features were selected based on domain knowledge and their statistical relationship with the target variable.

3.2 Methods Used

1. Correlation Thresholding:
   - Dropped features with low correlation to the target variable.
2. Recursive Feature Elimination (RFE):
   - Iteratively removed less impactful features.
3. Feature Importance (Tree-based Models):
   - Ranked features using Random Forest.
4. Mutual Information
5. Permutation Importance

3.3 Final Selection

The final feature set included:

- High-correlation variables like Category, Fat Content, and Visibility.
- Features ranked as important across multiple methods.

4 Feature Engineering

4.1 Transformations:

- Normalization/Standardization: Continuous variables were scaled using StandardScaler for models sensitive to feature magnitudes.
- Log Transformation: Applied to skewed variables to improve model interpretability and performance.
- Some custom transformers

4.2 Encoding:

- Categorical Variables: Encoded using one-hot encoding or label encoding, depending on the algorithm requirements.

4.3 Interaction Terms:

- New features representing interactions between key variables were created to capture non-linear relationships.

4.4 Missing Values:

- Imputed using mean, median, or mode strategies for numerical features and the most frequent category for categorical features.

5 Models and Hyperparameter Selection

5.1 Models Evaluated:

1. Linear Regression:
    - Hyperparameters tuning: Fit intercept set to false.
2. Gradient Boosting (CATBoost):
    - Hyperparameters tuning:

      'iterations': 664,

      'learning_rate': 0.0362387234017892,

      'depth': 3,

      'subsample': 0.7999456005681143,

      'colsample_bylevel': 0.7763198164104148,

      'l2_leaf_reg': 5.466280028000805,

      'random_strength': 1.1183292594718606,

      'bagging_temperature': 3.5375530112539995,

      'border_count': 105,

      'loss_function': 'MAE',

      'verbose': 0,

      smoothing = 0.4966048630439446

3. Gradient Boosting (XGBoost):
4. Hyperparameters tuned:

'n_estimators': 670,

'learning_rate': 0.06035467909986324,

'max_depth': 6,

'subsample': 0.923204157140139,

'colsample_bytree': 0.4883434932394659,

'reg_alpha': 5.410450954065299,

'reg_lambda': 1.1186894717633518,

'min_child_weight': 5,

'gamma': 2.3596898720558754,

'max_bin': 300,

'objective': 'reg:absoluteerror',

'eval_metric': 'mae',

'random_state': 42,

'enable_categorical': True

5. Support Vector Regressor (SVR):
   - Hyperparameters tuned

     'C': 2.6311460970378824,

     'epsilon': 0.35225148545949075,

     'kernel': 'poly',

     'degree': 2,

     'gamma': 'scale'

5.2 Hyperparameter Optimization:

- Optuna Search was used for efficient tuning.
- Cross-validation ensured robust evaluation of hyperparameter configurations.

5.3 Model Performance:

- Models were evaluated using metrics such as MAE.

- XGBoost and CatBoost consistently outperformed other models.

6 Conclusion:

The analysis and modeling process provided a comprehensive approach to developing and evaluating machine learning models. Feature engineering and hyperparameter significantly improved model performance, with XGBoost and CatBoost emerging as the most effective models.

7 Appendix

7.1 Visualization

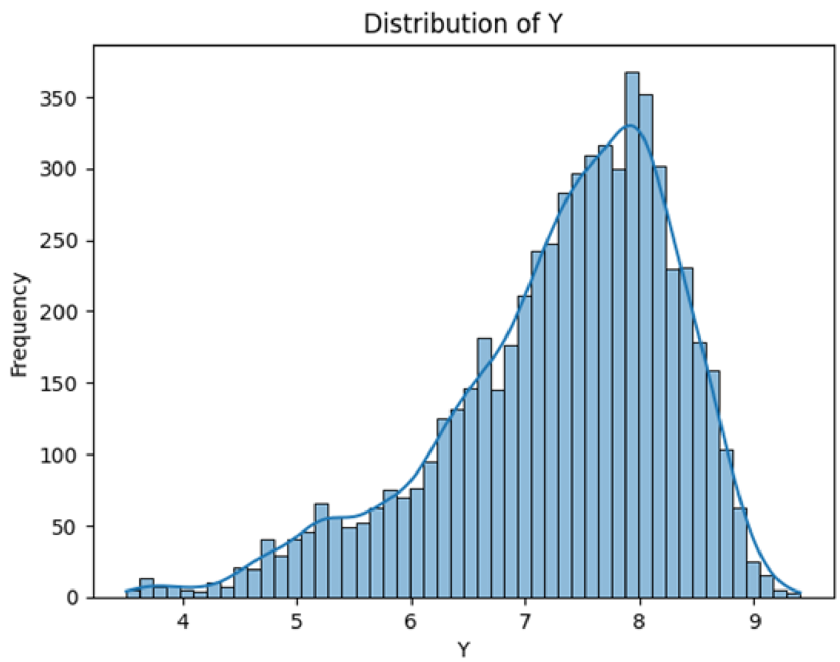


Figure 1 Correlation between Distribution and Frequency of Y

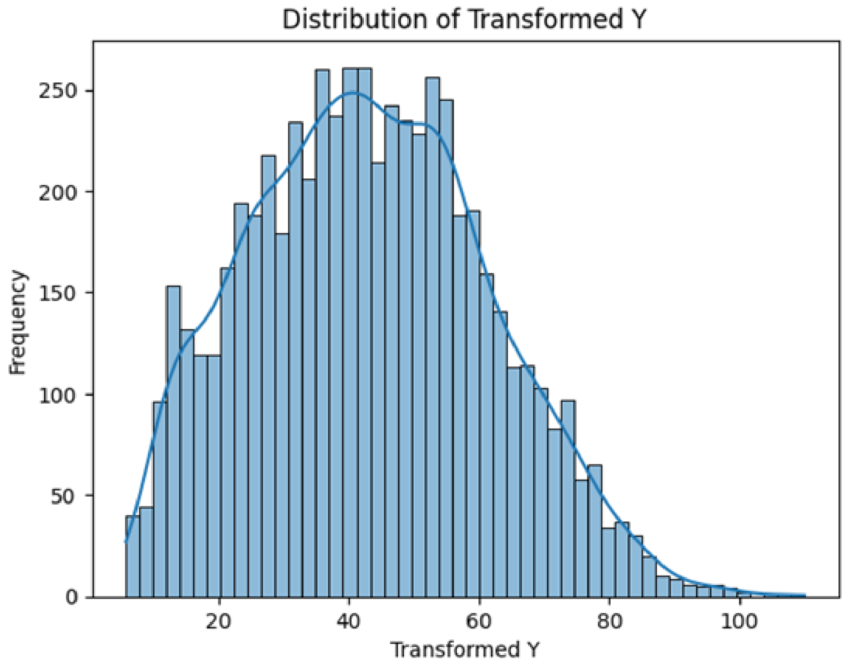


Figure 2 Correlation between Distribution and Frequency of Transformed Y

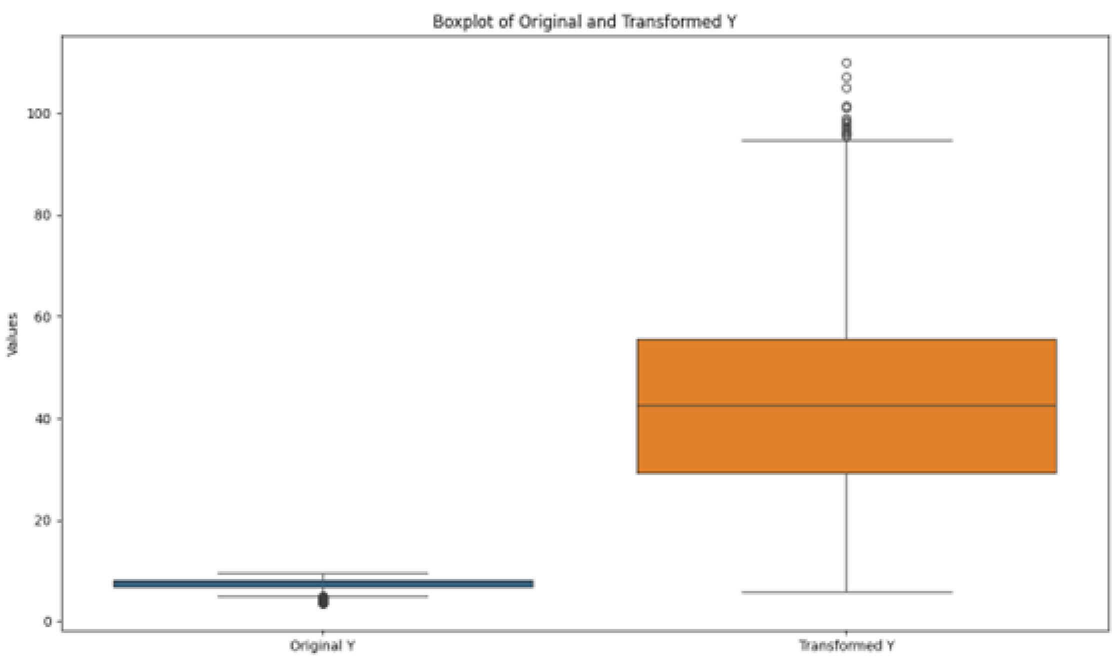


Figure 3 Boxplot of Original and Transformed Y

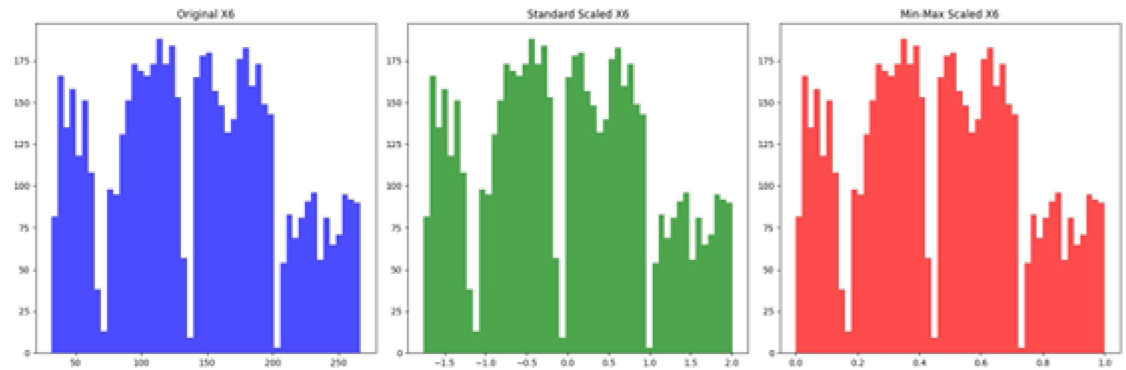


Figure 4 Apply Standard Scaler and Min-Max Scaler to the X6 column

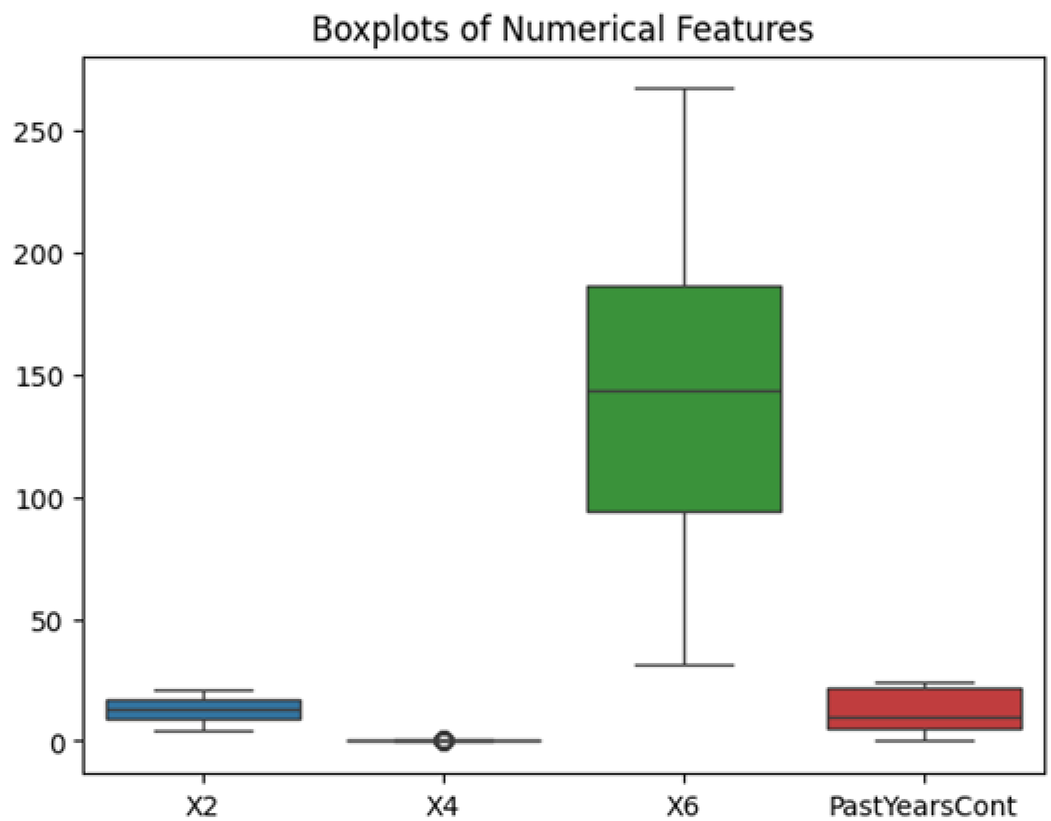
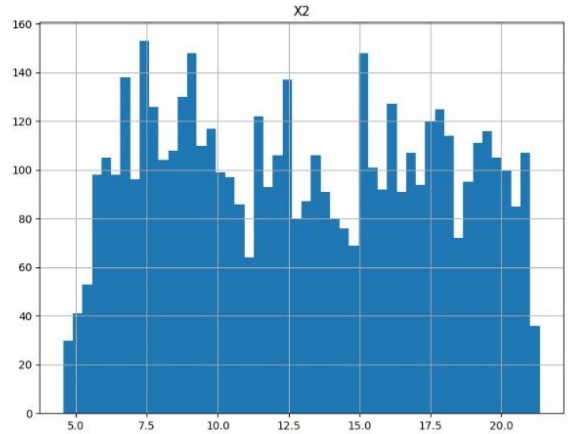
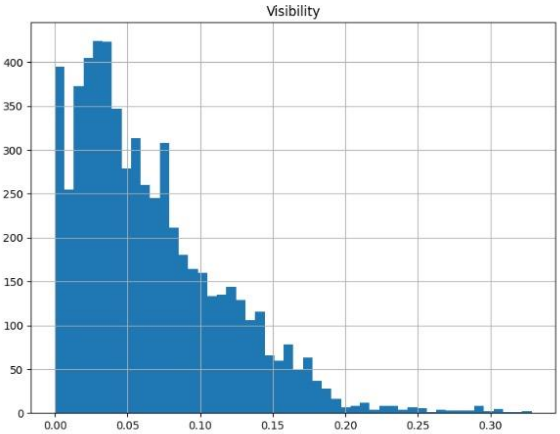
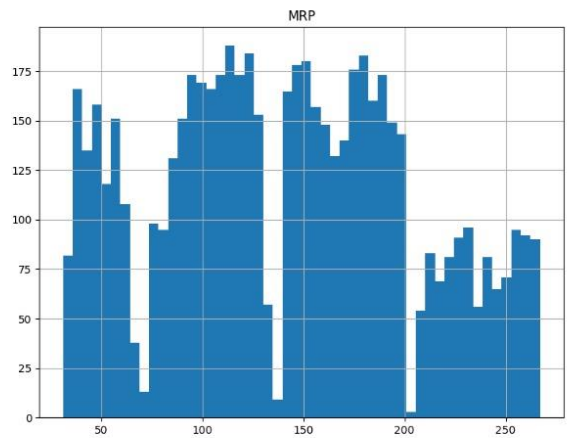
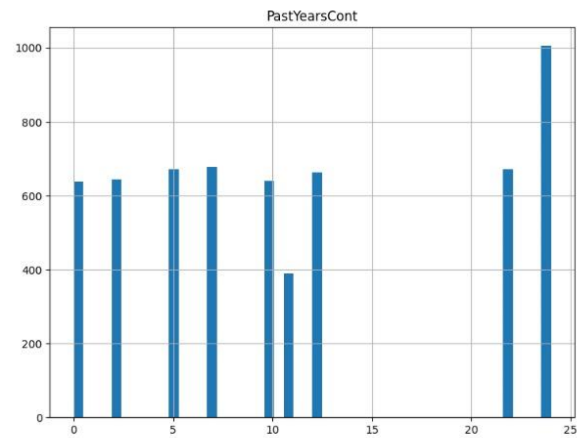


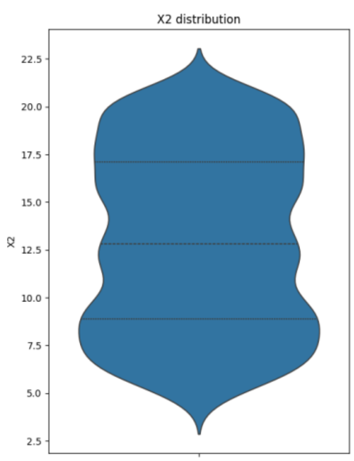
Figure 5 Overview of Numerical Features

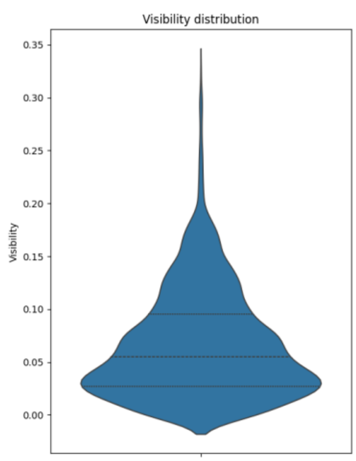

X2

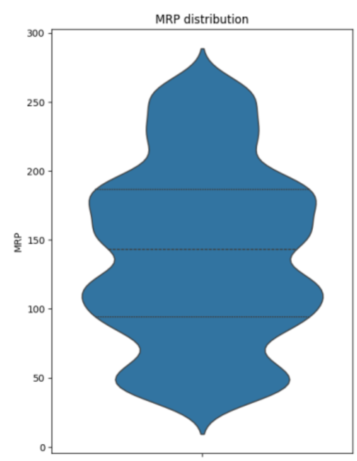

Visibility

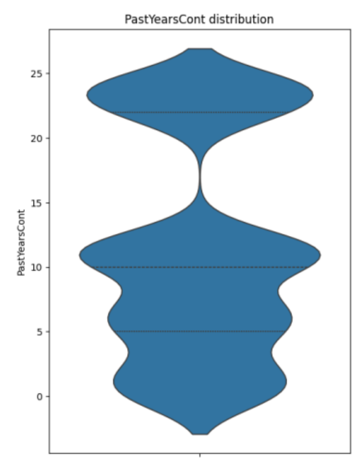

MRP

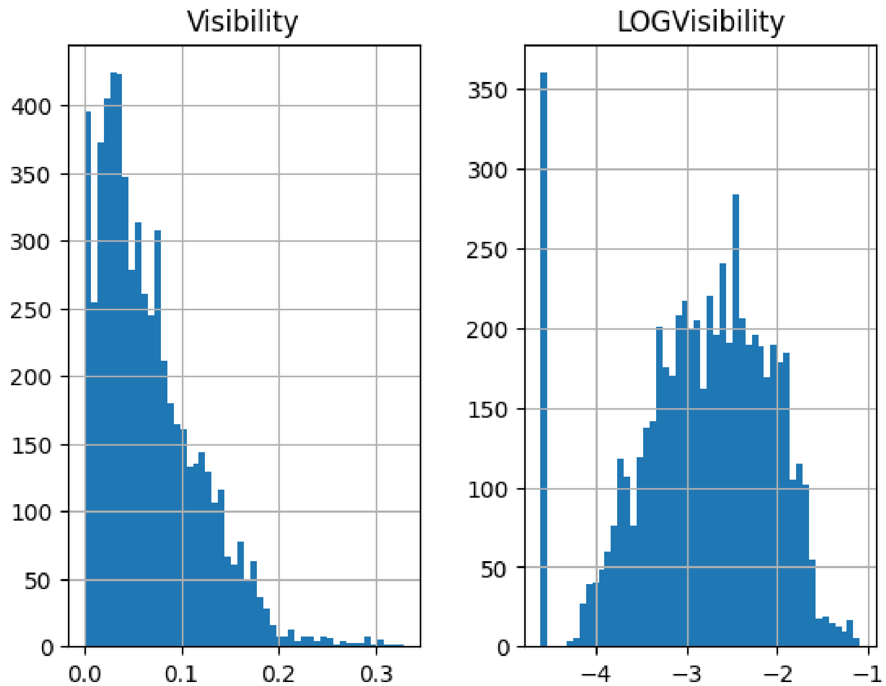

PastYearsCont


X2 distribution
X2


Visibility distribution
Visibility


MRP distribution
MRP


PastYearsCont distribution
PastYearsCont


Visibility
400
350
300
250
200
150
100
50
0
0.0
0.1
0.2
0.3
LOGVisibility
350
300
250
200
150
100
50
0
−4
−3
−2
−1

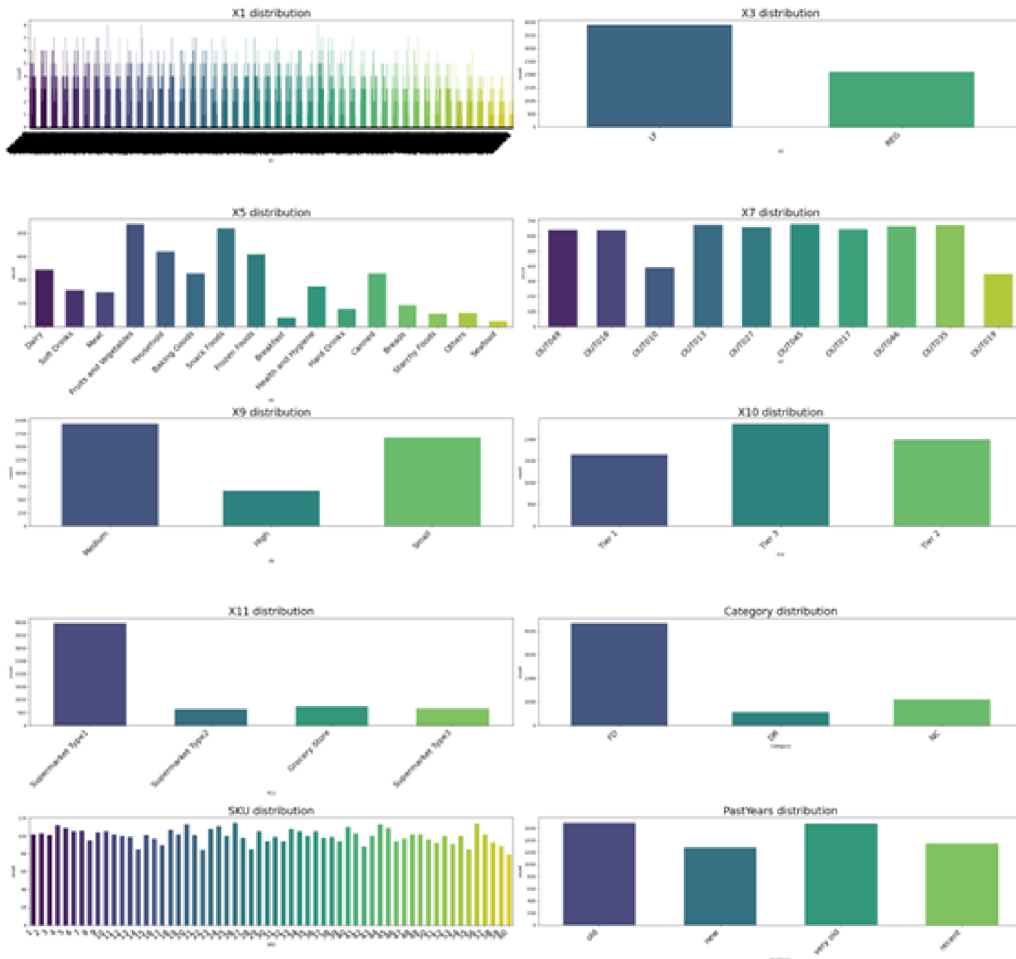


Figure 6 Overview of the all the features distributions

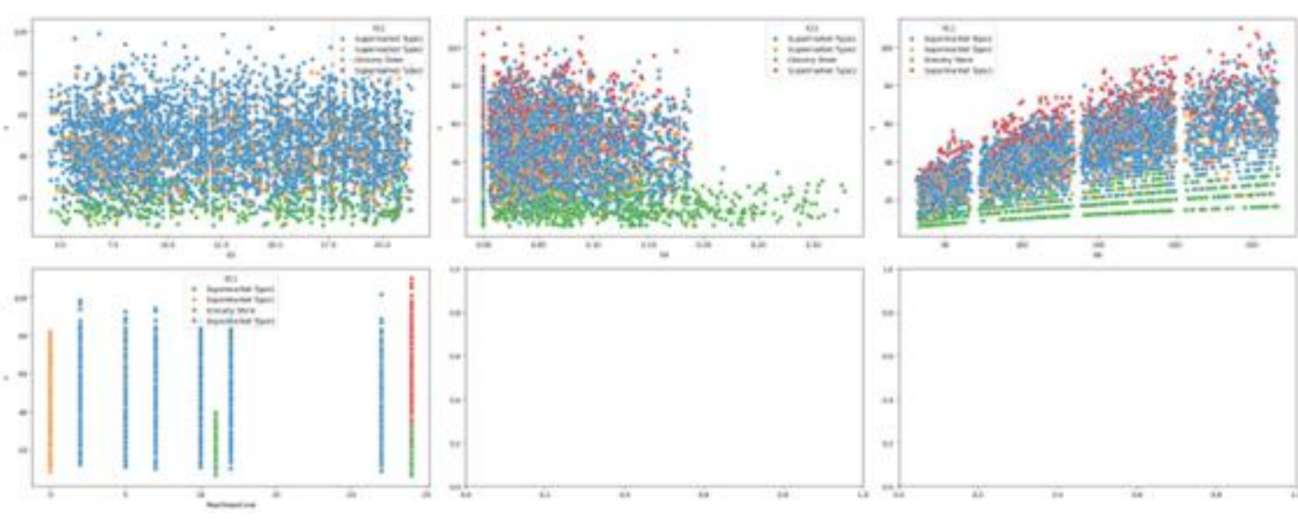
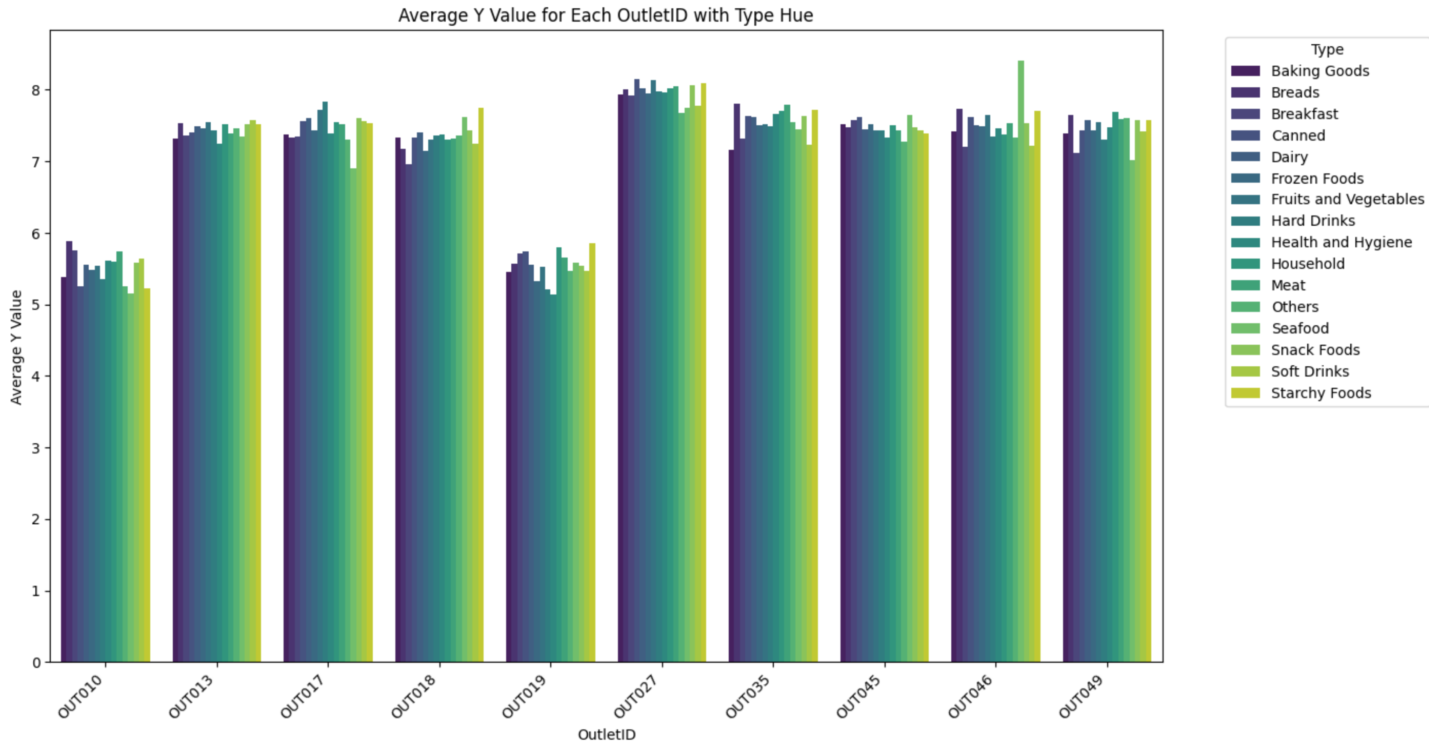
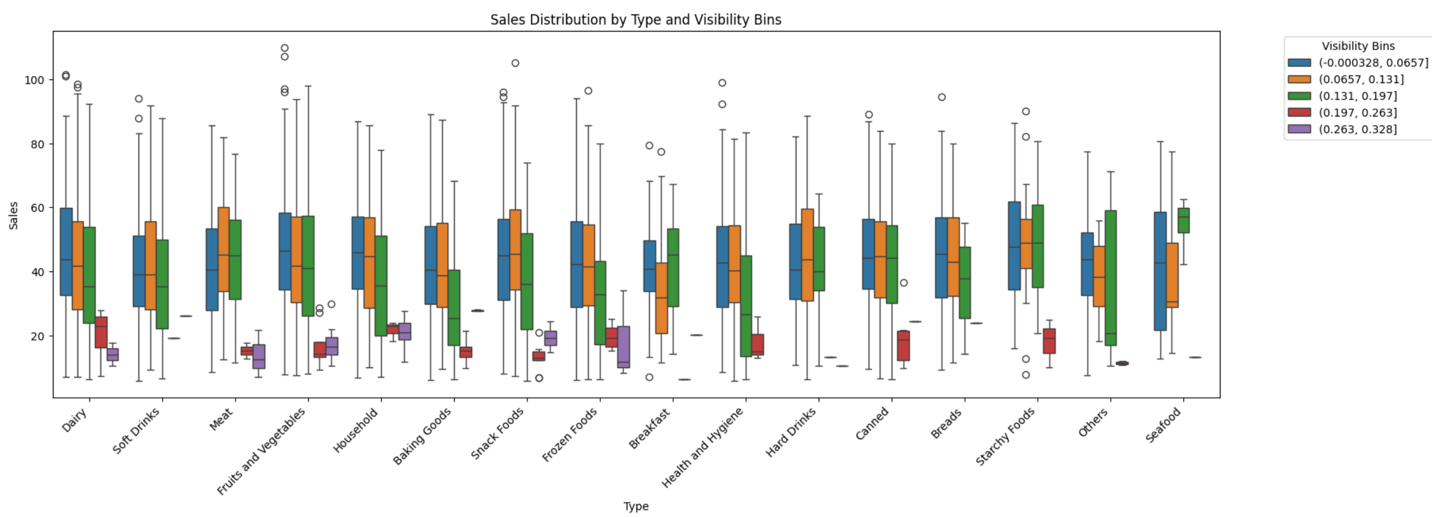


Figure 7 Correlations between features

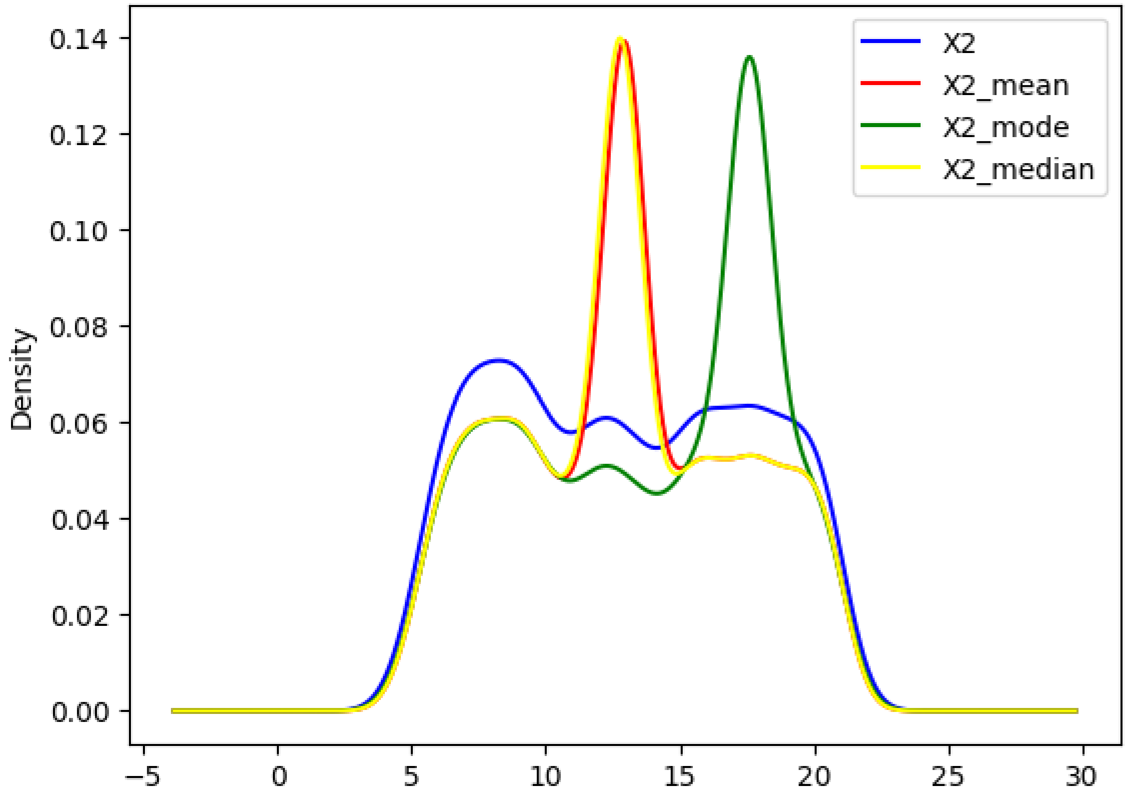

Average Y Value for Each OutletID with Type Hue
Average Y Value
OutletID
OUT010
OUT013
OUT017
OUT018
OUT019
OUT027
OUT035
OUT045
OUT046
OUT049
Type
Baking Goods
Breads
Breakfast
Canned
Dairy
Frozen Foods
Fruits and Vegetables
Hard Drinks
Health and Hygiene
Household
Meat
Others
Seafood
Snack Foods
Soft Drinks
Starchy Foods

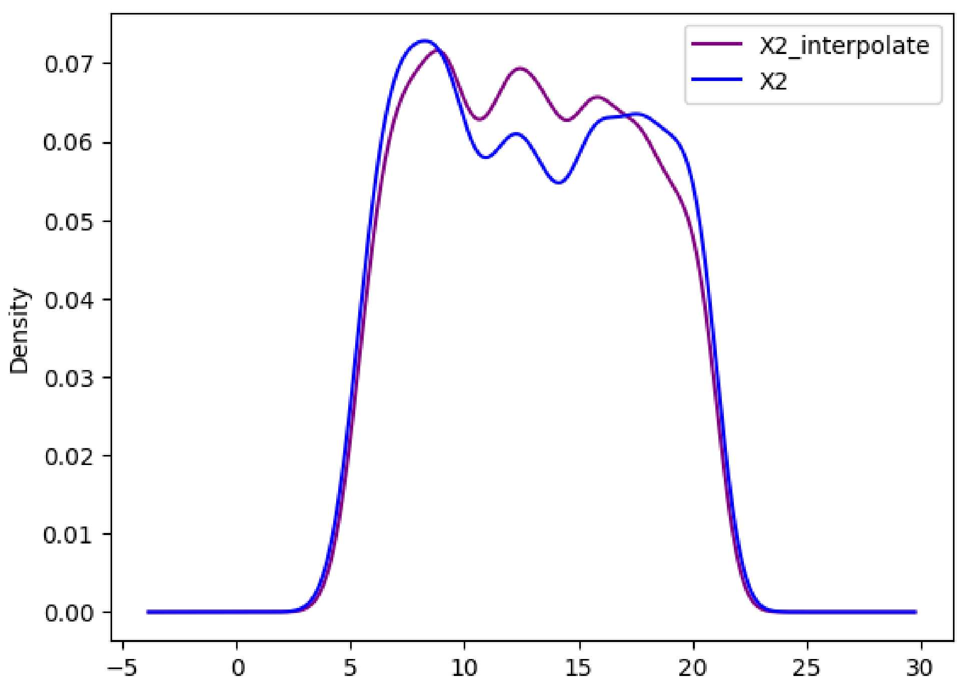

Sales Distribution by Type and Visibility Bins
Sales
Type
Dairy
Soft Drinks
Meat
Fruits and Vegetables
Household
Baking Goods
Snack Foods
Frozen Foods
Breakfast
Health and Hygiene
Hard Drinks
Canned
Breads
Starchy Foods
Others
Seafood
Visibility Bins
(-0.000328, 0.0657]
(0.0657, 0.131]
(0.131, 0.197]
(0.197, 0.263]
(0.263, 0.328]

Handling Missing Values:

Univariate Analysis:

X2 imputation techniques:





7.2 Code

7.2.1 CatBoost

```
class TypeMeanPriceTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, smoothing=0):
        self.smoothing = smoothing
        self.category_mean_price = None
        self.global_type_mean = None

    def fit(self, X, y=None):
        if y is None:
            raise ValueError("Target variable `y` cannot be None in the fit method.")

        data_ = X.copy()
        data_['Y'] = y

        # Global mean for each Type across all OutletTypes
        self.global_type_mean = data_.groupby('Type')['Y'].mean()

```

```python
        # Local mean and count for each (OutletType, Type)
        local_mean = data_.groupby(['Outlet_Type', 'Type'])['Y'].mean()
        local_count = data_.groupby(['Outlet_Type', 'Type'])['Y'].count()

        # Merge the global Type mean with the local statistics
        merged = local_mean.reset_index().merge(
            self.global_type_mean.reset_index(), on='Type', how='left', suffixes=('_local', '_global')
        )

        # Apply smoothing
        merged['TypeMeanPrice'] = (
            (merged['Y_local'] * local_count.values) +
            (merged['Y_global'] * self.smoothing)
        ) / (local_count.values + self.smoothing)

        self.category_mean_price = merged[['Outlet_Type', 'Type', 'TypeMeanPrice']]
        return self

    def transform(self, X):
        X = X.copy()
        # Merge the smoothed means back into the original data
        X = pd.merge(X, self.category_mean_price, on=['Outlet_Type', 'Type'], how='left')
        #X['TypeMeanPrice'] = np.log1p(X['TypeMeanPrice'])
        return X[['TypeMeanPrice']]

```

```python
class VisibilityZerosImputerZerosImputer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.item_Visibility_mean = defaultdict(lambda: None)  # Stores mean of Visibility per ItemID
        self.global_mean = None  # Fallback global mean for Visibility

    def fit(self, X, y=None):
        X = X.copy()

        # Ensure the required columns exist
        if 'ItemID' not in X.columns or 'Visibility' not in X.columns:
            raise ValueError("Both 'ItemID' and 'Visibility' columns must be present in the dataset.")

        # Ensure no None values in ItemID
        if X['ItemID'].isnull().any():
            raise ValueError("ItemID column contains None values.")

        # Calculate mean Visibility for each ItemID
        item_Visibility_mean = X.groupby('ItemID')['Visibility'].mean()
        self.item_Visibility_mean.update(item_Visibility_mean.to_dict())

        # Calculate global mean for the Visibility column
        if X['Visibility'].notnull().any():
            self.global_mean = X['Visibility'].mean()
        else:
            raise ValueError("Visibility column contains only NaN values.")

        return self

    def transform(self, X):
        X = X.copy()

        # Ensure the required columns exist
        if 'ItemID' not in X.columns or 'Visibility' not in X.columns:
```

```
            raise ValueError("Both 'ItemID' and 'Visibility' columns must be present in the dataset.")

        # Ensure no None values in ItemID
        if X['ItemID'].isnull().any():
            raise ValueError("ItemID column contains None values.")

        # Safely impute zero Visibilitys based on ItemID or global mean
        X['Visibility'] = X.apply(
            lambda row: self.item_Visibility_mean.get(row['ItemID'], self.global_mean)
            if row['Visibility'] == 0
            else row['Visibility'],
            axis=1
        )

        X['Visibility'] = X['Visibility'].replace(0, self.global_mean)
        X['Visibility'] = np.sqrt(X['Visibility'])

        return X[['Visibility']]

```

```
class X2NaNsImputer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.item_X2_mode = defaultdict(lambda: None)  # Stores mode of X2 per ItemID
        self.global_mean = None  # Fallback global mean for X2

    def fit(self, X, y=None):
        X = X.copy()
        # Ensure the required columns exist
        if 'ItemID' not in X.columns or 'X2' not in X.columns:
            raise ValueError("Both 'ItemID' and 'X2' columns must be present in the dataset.")

        # Ensure no None values in ItemID
        if X['ItemID'].isnull().any():
            raise ValueError("ItemID column contains None values.")

        # Calculate mode X2 for each ItemID
        item_X2_mode = X.groupby('ItemID')['X2'].agg(lambda x: x.mode().iloc[0] if not x.mode().empty else
np.nan)
        self.item_X2_mode.update(item_X2_mode.to_dict())

        # Calculate global mean for the X2 column
        if X['X2'].notnull().any():
            self.global_mean = X['X2'].mean()
        else:
            raise ValueError("X2 column contains only NaN values.")
        return self

    def transform(self, X):
        X = X.copy()

        # Ensure the required columns exist
        if 'ItemID' not in X.columns or 'X2' not in X.columns:
            raise ValueError("Both 'ItemID' and 'X2' columns must be present in the dataset.")

        # Ensure no None values in ItemID
        if X['ItemID'].isnull().any():
            raise ValueError("ItemID column contains None values.")

        # Safely impute NaN X2s based on ItemID or global mean
```

```
        X['X2'] = X.apply(
            lambda row: self.item_X2_mode.get(row['ItemID'], self.global_mean)
            if pd.isnull(row['X2'])
            else row['X2'],
            axis=1
        )

        X['X2'] = X['X2'].fillna(self.global_mean)
        X['X2'] = X['X2'].astype(float)

        return X[['X2']]

```

```
class KMeansClusterTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, num_clusters=5, random_state=42):
        """
        Parameters:
        - num_clusters: Number of clusters for KMeans.
        - random_state: Random state for reproducibility.
        """
        self.num_clusters = num_clusters
        self.random_state = random_state
        self.kmeans = None

    def fit(self, X, y=None):
        # Ensure X is a DataFrame
        if not isinstance(X, pd.DataFrame):
            X = pd.DataFrame(X, columns=['Feature1', 'Feature2'])

        # Fit the KMeans model
        self.kmeans = KMeans(n_clusters=self.num_clusters, random_state=self.random_state)
        self.kmeans.fit(X)
        return self

    def transform(self, X):
        # Ensure X is a DataFrame
        if not isinstance(X, pd.DataFrame):
            X = pd.DataFrame(X, columns=['Feature1', 'Feature2'])

        # Ensure the KMeans model is fitted
        if self.kmeans is None:
            raise ValueError("The KMeans model is not fitted yet. Please call 'fit' with appropriate arguments before using this method.")

        # Predict cluster labels
        cluster_labels = self.kmeans.predict(X)

        # Return only the cluster labels as a DataFrame
        return pd.DataFrame(cluster_labels, columns=X.columns, index=X.index)

```

```
class CustomFeatureTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, column_names):
        self.column_names = column_names
        # self.scaler_price_per_unit_weight = StandardScaler()
        # self.scaler_X2 = StandardScaler()
        # self.scaler_pricing_strategy = StandardScaler()
```

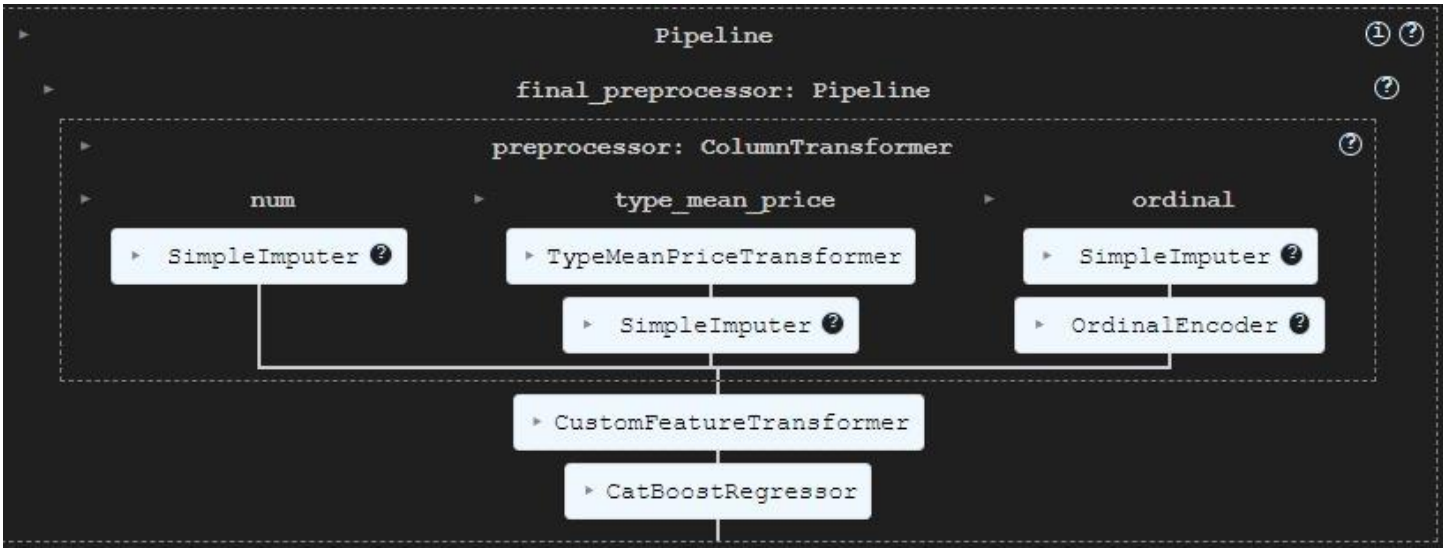
```

    def fit(self, X, y=None):
        # Convert to DataFrame with updated column names
        X = pd.DataFrame(X, columns=self.column_names)
        # Create new features
        # X['price_per_unit_weight'] = X['TypeMeanPrice_0'] / (X['X2'] + 0.0001)
        # X['pricing_strategy'] = X['MRP'] - (X['TypeMeanPrice_0'] * (X['Outlet_TypeOrdinal'] + 1) * (X['Location_Type'] + 1))

        # Fit the scalers on the training data
        # self.scaler_price_per_unit_weight.fit(X[['price_per_unit_weight']])
        # self.scaler_X2.fit(X[['X2']])
        # self.scaler_pricing_strategy.fit(X[['pricing_strategy']])

        return self

    def transform(self, X):
        # Convert to DataFrame with updated column names
        X = pd.DataFrame(X, columns=self.column_names)
        #X['OutletID'] = OutletID
       # X['OutletID'] = X['OutletID'].astype('category')

        #X['Total'] = Total_X
        # Convert object type features to float
        for col in X.select_dtypes(include=['object']).columns:
            X[col] = X[col].astype(float)
        # Create new features
        #X['BigMac_index2'] = (X['VisibilityCont'] + 0.0001) * (X['OutletType'] + 1) * (X['TypeMeanPrice'] + 3.24995) * (X['LocationType'] + 1)
        #X['price_per_unit_weight'] = X['TypeMeanPrice'] / (X['dummy'] + 0.0001)
        #X['BigMac_index2'] = np.sqrt(X['BigMac_index2'])
       # X['price_per_unit_weight'] = np.log1p(X['price_per_unit_weight'])
        #X['Visibility'] = X['Visibility'].rank(ascending=False)
        #X['ItemCategory'] = ItemCategory
        #X['OutletID'] = OutletID.astype(str)
        # X['MRP_Age_Interaction'] = X['MRP'] / (2010 - X['EstablishmentYear'])
        # X['MRP_Age_Interaction'] = np.log1p(X['MRP_Age_Interaction'])
        X.EstablishmentYear = X.EstablishmentYear.astype(str)
        X.MRP_cluster = X.MRP_cluster.astype(str)
        # X.Size = X.Size.astype(str)
        # X['ItemType'] = ItemType.astype(str)
        #X.ItemCategory = X.ItemCategory.astype('category')
        # X.FatContent = X.FatContent.astype(str)
        #X.X2 = X.X2.astype('category')
        #X.Visibility = X.Visibility.astype('category')
        #X.LocationType = X.LocationType.astype('category')
        # X['pricing_strategy'] = X['MRP'] - (X['TypeMeanPrice_0'] * (X['Outlet_TypeOrdinal'] + 1) * (X['Location_Type'] + 1))
        # X['RegionalPotential'] = X['Location_Type'] * (X['PastYearsCont'] + 1)
        # # Standard scale the new features
        # X['price_per_unit_weight'] = self.scaler_price_per_unit_weight.transform(X[['price_per_unit_weight']])
        # X['X2'] = self.scaler_X2.transform(X[['X2']])
        # X['pricing_strategy'] = self.scaler_pricing_strategy.transform(X[['pricing_strategy']])
        # Drop unwanted columns
        drop_columns = ['LocationType','Visibility','Size','FatContent','ItemType','MRP_Age_Interaction','EstablishmentYear']
        X.drop(columns=[col for col in drop_columns if col in X.columns], inplace=True)


```

```
        return X

```

```
class FrequencyImputer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.freqs = {}

    def fit(self, X, y=None):
        # Compute the frequency (count) of each category in each column
        if isinstance(X, pd.DataFrame):
            self.freqs = {col: X[col].value_counts() for col in X.columns}
        else:
            raise ValueError("Input must be a pandas DataFrame.")
        return self

    def transform(self, X):
        # Replace missing values with the frequency of each category
        X = X.copy()
        for col in X.columns:
            freq = self.freqs.get(col, None)
            if freq is not None:
                # Replace NaN values with the frequency count
                X[col] = X[col].apply(lambda x: freq[x] if pd.notna(x) else freq.idxmax() if pd.isna(x) else x)
        return X

```

```
from sklearn.pipeline import FunctionTransformer
all_features = [ 'EstablishmentYear', 'MRP',
'Visibility','TypeMeanPrice',"OutletType",'MRP_cluster','LocationType','FatContent','Size']
target_encoder_cols = ['Type']
numerical_cols = ['EstablishmentYear', 'MRP','Visibility']
VisibilityZeros_cols = ['ItemID','Visibility']
X2_imputer_cols = ['X2','ItemID']
ordinal_cols = [ "Outlet_Type",'MRP_cluster','Location_Type','FatContent','Size']
TypeMeanPriceTransformer_col = ['Outlet_Type' ,'Type']
freq_imputer = ['Type']  # only one-hot encoding
ordinal_categories = [  # FatContent
    ['Grocery Store', 'Supermarket Type1', 'Supermarket Type2', 'Supermarket Type3'],
    ['very low','low','high','very high'] ,
    ['Tier 1', 'Tier 3', 'Tier 2'],
    ['nofat','LF', 'REG'],
    ['Small', 'Medium', 'High']
   # Size
]
# Define frequency imputer pipeline
frequency_imputer_pipeline = Pipeline([
    ("freq_imputer", FrequencyImputer())
])
# Define numerical pipeline
numerical_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent"))
    #,("scaler", StandardScaler())
])

# size_imputer_pipeline = Pipeline([
#     ("custom_size_imputer", SizeImputer()),
#     ("imputer", SimpleImputer(strategy="most_frequent")),
```

```
#     ("ordinal", OrdinalEncoder(categories=[['Small', 'Medium', 'High']]))
# ])
# Define pipeline for X2 with X2 imputer and KMeans transformer
x2_pipeline = Pipeline([
    ("X2_imputer", X2NaNsImputer()),
    ("kmeans", KMeansClusterTransformer(num_clusters=5, random_state=42))
])
type_mean_price_pipeline = Pipeline([
    ("type_mean_price_transformer", TypeMeanPriceTransformer(smoothing=0.5)),
    ("imputer", SimpleImputer(strategy="mean"))
])
# Define one-hot encoding pipeline
ordinal_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("ordinal", OrdinalEncoder(categories=ordinal_categories))
])
# Define one-hot encoding pipeline for onehot-transform columns
onehot_transform_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(categories=[['Snack Foods', 'Frozen Foods', 'Fruits and Vegetables', 'Canned']], handle_unknown="ignore"))
])
# Define target encoding pipeline
target_encoding_pipeline = Pipeline([
    ("target_encoder", TargetEncoder(cols=target_encoder_cols , smoothing=0.5) )
])
# Combine all pipelines into a ColumnTransformer
preprocessor = ColumnTransformer([
    #("X2_imputer", X2NaNsImputer(), X2_imputer_cols),
    #("kmeans_X2", KMeansClusterTransformer(num_clusters=5, random_state=42), ['X2']),
    #("kmeans_visibility", KMeansClusterTransformer(num_clusters=5, random_state=42), ['Visibility']),
    ("num", numerical_pipeline, numerical_cols),
    ("type_mean_price", type_mean_price_pipeline, TypeMeanPriceTransformer_col),
    ("ordinal", ordinal_pipeline, ordinal_cols)
    #("freq_imputer", frequency_imputer_pipeline, ['TypeFreq'])
])
# Define the final pipeline
# Create the extended pipeline
final_pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("custom_features", CustomFeatureTransformer(column_names=all_features))
    #("kmeans", KMeansTransformer(num_clusters=5, random_state=42))
])
```




```python
import optuna
from catboost import CatBoostRegressor
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

def objective_catboost(trial):
    param = {
        'iterations': trial.suggest_int('iterations', 100, 800),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.1),
        'depth': trial.suggest_int('depth', 3, 6),
        'subsample': trial.suggest_float('subsample', 0.5, 0.9),
        'colsample_bylevel': trial.suggest_float('colsample_bylevel', 0.5, 0.9),
        'l2_leaf_reg': trial.suggest_float('l2_leaf_reg', 1, 10),
        'random_strength': trial.suggest_float('random_strength', 1, 5),
        'bagging_temperature': trial.suggest_float('bagging_temperature', 1, 5),
        'border_count': trial.suggest_int('border_count', 32, 128),
        'loss_function': 'MAE'
    }

    # Convert categorical columns to string
    # X_trans_catboost = X_trans.copy()
    # for col in X_trans_catboost.select_dtypes(include=['category']).columns:
    #     X_trans_catboost[col] = X_trans_catboost[col].astype(str)
    smoothing = trial.suggest_float('smoothing', 0.1, 5)
    preprocessor.set_params(
        type_mean_price__type_mean_price_transformer__smoothing=smoothing
    )
    # Create a pipeline with the final transformer and CatBoost regressor
```

```
    model = CatBoostRegressor(**param, verbose=0, cat_features=['MRP_cluster'], early_stopping_rounds=50)
    pipeline = Pipeline([
        ('final_transform', final_pipeline),
        ('catboost', model)
    ])

    # Perform cross-validation
    scores = cross_val_score(pipeline, X, Y, cv=5, scoring='neg_mean_absolute_error')
    mae = -scores.mean()
    return mae

# Create an Optuna study and optimize the objective function
study_catboost = optuna.create_study(direction='minimize')
study_catboost.optimize(objective_catboost, n_trials=10)

# Print the best parameters and MAE
print("CatBoost Best Parameters:", study_catboost.best_params)
print("CatBoost Best MAE:", study_catboost.best_value)

```

```
# Fit the pipeline
cat_pipeline.fit(X_train, Y_train)

# Transform the training data
X_train_transformed = cat_pipeline.named_steps['final_preprocessor'].transform(X_train)

# Calculate feature importances
mdi_importances = pd.Series(cat_pipeline.named_steps['catboost'].feature_importances_,
index=X_train_transformed.columns)
tree_importance_sorted_idx = np.argsort(cat_pipeline.named_steps['catboost'].feature_importances_)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 8))

# Plot Gini importance
mdi_importances.sort_values().plot.barh(ax=ax1)
ax1.set_xlabel("Gini importance")

# Plot permutation importance
plot_permutation_importance(cat_pipeline.named_steps['catboost'], X_train_transformed, Y_train, ax2)
ax2.set_xlabel("Decrease in accuracy score")

fig.suptitle("Impurity-based vs. permutation importances on multicollinear features (train set)")
fig.tight_layout()
plt.show()

```

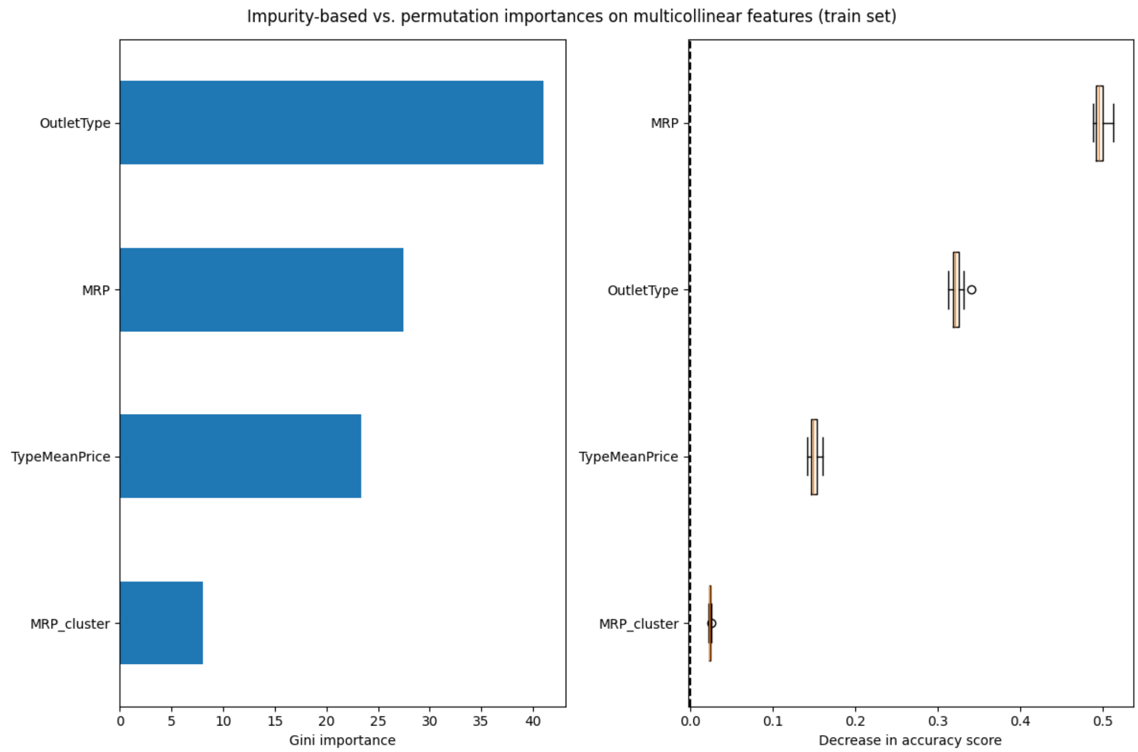


Figure 8 Impurity-based vs. permutation importances on multicollinear features

7.3 Linear Model

```python
class ItemWeightImputer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.item_weight_mode = defaultdict(lambda: None)  # Stores mode of Item_Weight per Item_Identifier
        self.global_mean = None  # Fallback global mean for Item_Weight

    def fit(self, X, y=None):
        X = X.copy()

        # Ensure the required columns exist
        if 'Item_Identifier' not in X.columns or 'Item_Weight' not in X.columns:
            raise ValueError("Both 'Item_Identifier' and 'Item_Weight' columns must be present in the dataset.")

        # Ensure no None values in Item_Identifier
        if X['Item_Identifier'].isnull().any():
            raise ValueError("Item_Identifier column contains None values.")

        # Calculate mode Item_Weight for each Item_Identifier
        item_weight_mode = X.groupby('Item_Identifier')['Item_Weight'].agg(lambda x: x.mode().iloc[0] if not x.mode().empty else np.nan)
        self.item_weight_mode.update(item_weight_mode.to_dict())

        # Calculate global mean for the Item_Weight column
        if X['Item_Weight'].notnull().any():
            self.global_mean = X['Item_Weight'].mean()
        else:
            raise ValueError("Item_Weight column contains only NaN values.")

        return self

    def transform(self, X):
        X = X.copy()

        # Ensure the required columns exist
        if 'Item_Identifier' not in X.columns or 'Item_Weight' not in X.columns:
            raise ValueError("Both 'Item_Identifier' and 'Item_Weight' columns must be present in the dataset.")

        # Ensure no None values in Item_Identifier
        if X['Item_Identifier'].isnull().any():
            raise ValueError("Item_Identifier column contains None values.")

        # Safely impute NaN Item_Weights based on Item_Identifier or global mean
        X['Item_Weight'] = X.apply(
            lambda row: self.item_weight_mode.get(row['Item_Identifier'], self.global_mean)
            if pd.isnull(row['Item_Weight'])
            else row['Item_Weight'],
            axis=1
        )

        X['Item_Weight'] = X['Item_Weight'].fillna(self.global_mean)

        return X[['Item_Weight']]
```

```
X['Category'] = X['Item_Identifier'].str[:2]
X["Item_Fat_Content"] = X["Item_Fat_Content"].replace({"low fat": "LF", "Low Fat": "LF", "Regular": "REG",
"reg": "REG"})
X.loc[X['Category'] == 'NC', 'Item_Fat_Content'] = 'nofat'
impute_sizes = {
    "OUT010": "Small",
    "OUT017": "Small",
    "OUT045": "Medium"
}
X['Outlet_Size'] = X.apply(
    lambda row: impute_sizes[row['Outlet_Identifier']] if pd.isnull(row['Outlet_Size']) else
row['Outlet_Size'], axis=1
)
X['MRP_cluster']=pd.cut(X["Item_MRP"],bins=[25,69,137,203,270],labels=['very low','low','high','very
high'],right=True)
weightimputer = ItemWeightImputer()
X["Item_Weight"] = weightimputer.fit_transform(X)
X["Weight_per_Unit_MRP"] = X["Item_Weight"]/X["Item_MRP"]

```

```
#target_encoder_cols = ['Item_Type']
numerical_cols = ["Item_Weight","Weight_per_Unit_MRP","Item_Visibility"]
# ordinal_cols =["Outlet_Size"]
one_hot_columns = ["Outlet_Location_Type","Category","Item_Fat_Content","Outlet_Size","Outlet_Type"]
target_encoder_cols = ["Item_Identifier",'Item_Type','Outlet_Identifier']
ordinal_categories = [['very low', 'low', 'high', 'very high']]
# Numerical pipeline
# numerical_pipeline = Pipeline([
#     ("group_mean_imputer", FillNaWithGroupMode(group_col="Item_Identifier", target_col="Item_Weight")),
#     ("mean_imputer", SimpleImputer(strategy="mean"))
#     # ("scaler", StandardScaler())
# ])
numerical_pipeline = Pipeline([
    # ("group_mode_imputer", FillNaWithGroupMode(group_col=group_col, target_col='Item_Weight')),
    ("mean_imputer", SimpleImputer(strategy='mean')),
    ("scaler", StandardScaler())
])

#Ordinal pipeline
# ordinal_pipeline = Pipeline([
#     ("imputer", SimpleImputer(strategy="most_frequent")),
#     ("ordinal", OrdinalEncoder(categories=ordinal_categories))
# ])
target_encoding_pipeline = Pipeline([
    ("target_encoder", TargetEncoder(cols=target_encoder_cols , smoothing=0.5) )
])
# One-hot encoding pipeline
onehot_transform_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown="ignore",sparse_output=False))
])
ordinal_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("ordinal", OrdinalEncoder(categories=ordinal_categories))
])

# Target encoding pipeline
# target_encoding_pipeline = Pipeline([
```

```
#     ("target_encoder", TargetEncoder(cols=target_encoder_cols))
# ])

# Combine all pipelines into a ColumnTransformer
preprocessor = ColumnTransformer([
    ("num", numerical_pipeline, numerical_cols),
    ("target_encoder", target_encoding_pipeline, target_encoder_cols),
    ("onehot", onehot_transform_pipeline, one_hot_columns),
    ("ordinal", ordinal_pipeline, ["MRP_cluster"])

])

# Final pipeline
final_pipeline = Pipeline([
    ("preprocessor", preprocessor),
])

```

```
import optuna
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

def objective_linear_regression(trial):
    # LinearRegression does not have many hyperparameters to tune, but we can still create a pipeline
    param = {
        'fit_intercept': trial.suggest_categorical('fit_intercept', [True, False])
    }

    # Create a pipeline with the final transformer and LinearRegression
    model = LinearRegression(**param)
    pipeline = Pipeline([
        ('final_transform', final_pipeline),
        ('linear_regression', model)
    ])

    # Perform cross-validation with error handling
    try:
        scores = cross_val_score(pipeline, X, Y, cv=5, scoring='neg_mean_absolute_error', error_score='raise')
        mae = -scores.mean()
        print(f"Trial MAE: {mae}")
    except Exception as e:
        print(f"Error during cross-validation: {e}")
        mae = float('inf')  # Assign a high error value if an exception occurs

    return mae

study_linear_regression = optuna.create_study(direction='minimize')
study_linear_regression.optimize(objective_linear_regression, n_trials=50)

print("Linear Regression Best Parameters:", study_linear_regression.best_params)
print("Linear Regression Best MAE:", study_linear_regression.best_value)

# Extract the best model
best_params = study_linear_regression.best_params
best_model = LinearRegression(**best_params)
best_pipeline = Pipeline([
    ('final_transform', final_pipeline),
    ('linear_regression', best_model)
```

```
])

# Fit the best pipeline on the entire dataset
best_pipeline.fit(X, Y)

# Now you can use best_pipeline for predictions
predictions = best_pipeline.predict(X)
print(predictions)

```

```
# Get the feature names from the preprocessor
feature_names = final_pipeline.named_steps['preprocessor'].get_feature_names_out()

# Get the coefficients from the linear regression model
coefficients = best_model.coef_

# Create a DataFrame to hold feature names and their corresponding coefficients
feature_importance = pd.DataFrame({
    'Feature': feature_names,
    'Coefficient': coefficients
})

# Sort the DataFrame by the absolute value of the coefficients in descending order
feature_importance['Absolute_Coefficient'] = feature_importance['Coefficient'].abs()
feature_importance = feature_importance.sort_values(by='Absolute_Coefficient', ascending=False)

# Print the top 10 most important features
print(feature_importance.head(10))

```


```
# Get the feature names from the preprocessor
feature_names = final_pipeline.named_steps['preprocessor'].get_feature_names_out()

# Get the coefficients from the linear regression model
coefficients = best_model.coef_

# Create a DataFrame to hold feature names and their corresponding coefficients
feature_importance = pd.DataFrame({
    'Feature': feature_names,
    'Coefficient': coefficients
})

# Sort the DataFrame by the absolute value of the coefficients in descending order
feature_importance['Absolute_Coefficient'] = feature_importance['Coefficient'].abs()
feature_importance = feature_importance.sort_values(by='Absolute_Coefficient', ascending=False)

# Print the top 10 most important features
print(feature_importance.head(10))

```


```
Feature   Coefficient  Absolute_Coefficient
8   onehot__Outlet_Location_Type_Tier 3 -8.764891e+09          8.764891e+09
7   onehot__Outlet_Location_Type_Tier 2 -8.764891e+09          8.764891e+09
6   onehot__Outlet_Location_Type_Tier 1 -8.764891e+09          8.764891e+09
12          onehot__Item_Fat_Content_LF -8.764735e+09          8.764735e+09
```

```
13          onehot__Item_Fat_Content_REG -8.764735e+09          8.764735e+09
11                   onehot__Category_NC -8.764199e+09          8.764199e+09
14        onehot__Item_Fat_Content_nofat -8.763889e+09          8.763889e+09
10                   onehot__Category_FD -8.763352e+09          8.763352e+09
9                    onehot__Category_DR -8.763352e+09          8.763352e+09
17              onehot__Outlet_Size_Small -8.762929e+09          8.762929e+09

```

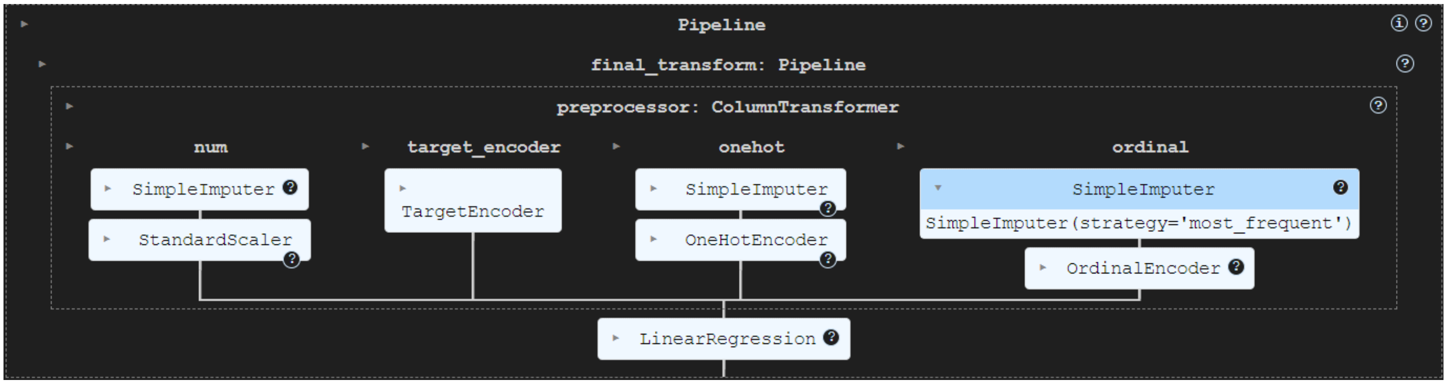


Figure 9 Linear Regression Pipeline

7.4 SVR Model

```python
class ItemWeightImputer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.item_weight_mode = defaultdict(lambda: None)  # Stores mode of Item_Weight per Item_Identifier
        self.global_mean = None  # Fallback global mean for Item_Weight

    def fit(self, X, y=None):
        X = X.copy()

        # Ensure the required columns exist
        if 'Item_Identifier' not in X.columns or 'Item_Weight' not in X.columns:
            raise ValueError("Both 'Item_Identifier' and 'Item_Weight' columns must be present in the dataset.")

        # Ensure no None values in Item_Identifier
        if X['Item_Identifier'].isnull().any():
            raise ValueError("Item_Identifier column contains None values.")

        # Calculate mode Item_Weight for each Item_Identifier
        item_weight_mode = X.groupby('Item_Identifier')['Item_Weight'].agg(lambda x: x.mode().iloc[0] if not x.mode().empty else np.nan)
        self.item_weight_mode.update(item_weight_mode.to_dict())

        # Calculate global mean for the Item_Weight column
        if X['Item_Weight'].notnull().any():
            self.global_mean = X['Item_Weight'].mean()
        else:
            raise ValueError("Item_Weight column contains only NaN values.")

        return self

```

```
    def transform(self, X):
        X = X.copy()

        # Ensure the required columns exist
        if 'Item_Identifier' not in X.columns or 'Item_Weight' not in X.columns:
            raise ValueError("Both 'Item_Identifier' and 'Item_Weight' columns must be present in the dataset.")

        # Ensure no None values in Item_Identifier
        if X['Item_Identifier'].isnull().any():
            raise ValueError("Item_Identifier column contains None values.")

        # Safely impute NaN Item_Weights based on Item_Identifier or global mean
        X['Item_Weight'] = X.apply(
            lambda row: self.item_weight_mode.get(row['Item_Identifier'], self.global_mean)
            if pd.isnull(row['Item_Weight'])
            else row['Item_Weight'],
            axis=1
        )

        X['Item_Weight'] = X['Item_Weight'].fillna(self.global_mean)

        return X[['Item_Weight']]

```

```
class VisibilityZerosImputer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.item_visibility_mean = defaultdict(lambda: None)  # Stores mean of Item_Visibility per Item_Identifier
        self.global_mean = None  # Fallback global mean for Item_Visibility

    def fit(self, X, y=None):
        X = X.copy()

        # Ensure the required columns exist
        if 'Item_Identifier' not in X.columns or 'Item_Visibility' not in X.columns:
            raise ValueError("Both 'Item_Identifier' and 'Item_Visibility' columns must be present in the dataset.")

        # Ensure no None values in Item_Identifier
        if X['Item_Identifier'].isnull().any():
            raise ValueError("Item_Identifier column contains None values.")

        # Calculate mean Item_Visibility for each Item_Identifier
        item_visibility_mean = X.groupby('Item_Identifier')['Item_Visibility'].mean()
        self.item_visibility_mean.update(item_visibility_mean.to_dict())

        # Calculate global mean for the Item_Visibility column
        if X['Item_Visibility'].notnull().any():
            self.global_mean = X['Item_Visibility'].mean()
        else:
            raise ValueError("Item_Visibility column contains only NaN values.")

        return self

    def transform(self, X):
        X = X.copy()

        # Ensure the required columns exist
```

```
        if 'Item_Identifier' not in X.columns or 'Item_Visibility' not in X.columns:
            raise ValueError("Both 'Item_Identifier' and 'Item_Visibility' columns must be present in the dataset.")

        # Ensure no None values in Item_Identifier
        if X['Item_Identifier'].isnull().any():
            raise ValueError("Item_Identifier column contains None values.")

        # Safely impute zero Item_Visibility based on Item_Identifier or global mean
        X['Item_Visibility'] = X.apply(
            lambda row: self.item_visibility_mean.get(row['Item_Identifier'], self.global_mean)
            if row['Item_Visibility'] == 0
            else row['Item_Visibility'],
            axis=1
        )

        X['Item_Visibility'] = X['Item_Visibility'].replace(0, self.global_mean)
        X['Item_Visibility'] = np.sqrt(X['Item_Visibility'])

        return X[['Item_Visibility']]

```

```
X['Item_Type_Outlet_Type'] = X['Item_Type'] + "_" + X['Outlet_Type']
X['Item_Fat_Content_Outlet_Type'] = X['Item_Fat_Content'] + "_" + X['Outlet_Type']
X['Outlet_Size_Outlet_Location_Type'] = X['Outlet_Size'] + "_" + X['Outlet_Location_Type']
X['Item_Type_Item_Fat_Content'] = X['Item_Type'] + "_" + X['Item_Fat_Content']
X['Outlet_Type_Outlet_Location_Type'] = X['Outlet_Type'] + "_" + X['Outlet_Location_Type']

# Numerical × Numerical Interactions
X['Item_MRP_Outlet_Establishment_Year'] = X['Item_MRP'] * X['Outlet_Establishment_Year']
X['Item_MRP_Item_Visibility'] = X['Item_MRP'] * X['Item_Visibility']
X['Years_Operating_Outlet_Type'] = X['age'] * X['Outlet_Type'].apply(lambda x: hash(x) % 1000)
X['Outlet_Identifier_Years_Operating'] = X['Outlet_Identifier'].apply(lambda x: hash(x) % 1000) * X['age']

# Numerical × Categorical Interactions
X['Item_Visibility_Item_Type'] = X['Item_Visibility'] * X['Item_Type'].apply(lambda x: hash(x) % 1000)

```

```
# Define the columns for each type of transformation
numerical_cols = ["Item_Weight", "Weight_per_Unit_MRP", "Item_Visibility", "Item_MRP_Outlet_Establishment_Year", "Item_MRP_Item_Visibility", "Years_Operating_Outlet_Type", "Outlet_Identifier_Years_Operating", "Item_Visibility_Item_Type"]
one_hot_columns = ["Outlet_Location_Type", "Category", "Item_Fat_Content", "Outlet_Size", "Outlet_Type", "Item_Type_Outlet_Type", "Item_Fat_Content_Outlet_Type", "Outlet_Size_Outlet_Location_Type", "Item_Type_Item_Fat_Content", "Outlet_Type_Outlet_Location_Type"]
target_encoder_cols = ['Item_Identifier','Item_Type', 'Outlet_Identifier']
ordinal_categories = [['very low', 'low', 'high', 'very high']]

# Numerical pipeline
numerical_pipeline = Pipeline([
    ("mean_imputer", SimpleImputer(strategy='mean')),
    ("scaler", StandardScaler())
])

#Target encoding pipeline
target_encoding_pipeline = Pipeline([
    ("target_encoder", TargetEncoder(cols=target_encoder_cols, smoothing=0.5))
```

```
])

# One-hot encoding pipeline
onehot_transform_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown="ignore", sparse_output=False))
])

# Ordinal encoding pipeline
ordinal_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("ordinal", OrdinalEncoder(categories=ordinal_categories))
])

# Combine all pipelines into a ColumnTransformer
preprocessor = ColumnTransformer([
    ("num", numerical_pipeline, numerical_cols),
    ("target_encoder", target_encoding_pipeline, target_encoder_cols),
    ("onehot", onehot_transform_pipeline, one_hot_columns),
    ("ordinal", ordinal_pipeline, ["MRP_cluster"])
])

# Final pipeline
final_pipeline = Pipeline([
    ("preprocessor", preprocessor)
```

```
import optuna
from sklearn.svm import SVR
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

def objective_svr(trial):
    param = {
        'C': trial.suggest_float('C', 0.1, 100.0),
        'epsilon': trial.suggest_float('epsilon', 0.01, 1.0),
        'kernel': trial.suggest_categorical('kernel', ['linear', 'poly', 'rbf', 'sigmoid']),
        'degree': trial.suggest_int('degree', 2, 5) if trial.suggest_categorical('kernel', ['linear', 'poly', 'rbf', 'sigmoid']) == 'poly' else 3,
        'gamma': trial.suggest_categorical('gamma', ['scale', 'auto'])
    }

    # Create a pipeline with the final transformer and SVR
    model = SVR(**param)
    pipeline = Pipeline([
        ('final_transform', final_pipeline),
        ('svr', model)
    ])

    # Perform cross-validation with error handling
    try:
        scores = cross_val_score(pipeline, X, Y, cv=5, scoring='neg_mean_absolute_error', error_score='raise')
        mae = -scores.mean()
        print(f"Trial MAE: {mae}")
    except Exception as e:
        print(f"Error during cross-validation: {e}")
        mae = float('inf')  # Assign a high error value if an exception occurs

    return mae

```

```python
study_svr = optuna.create_study(direction='minimize')
study_svr.optimize(objective_svr, n_trials=50)

print("SVR Best Parameters:", study_svr.best_params)
print("SVR Best MAE:", study_svr.best_value)
```

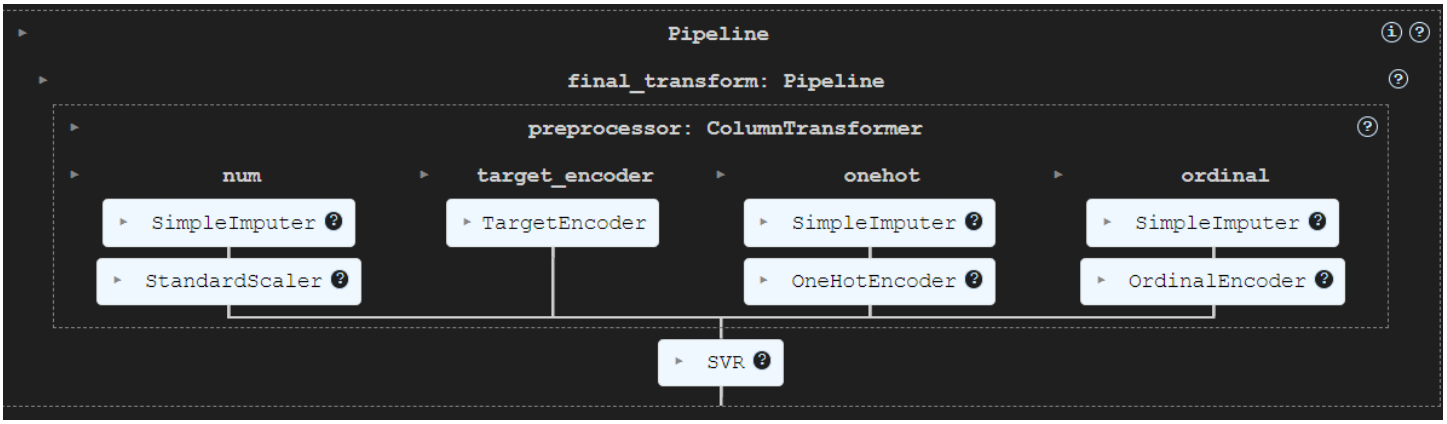


Figure 10 SVR Pipeline

7.5 XGBoost Stacking Model

```
class TypeMeanPriceTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, smoothing=1):
        self.smoothing = smoothing
        self.category_mean_price = None

    def fit(self, X, y=None):
        if y is None:
            raise ValueError("Target variable `y` cannot be None in the fit method.")
        data_ = X.copy()
        data_['Y'] = y
        # Calculate the mean price for each OutletID and Type
        mean_price = data_.groupby(['OutletID', 'Type'])['Y'].mean()
        count = data_.groupby(['OutletID', 'Type'])['Y'].count()
        global_mean = data_['Y'].mean()

        # Apply smoothing
        self.category_mean_price = ((mean_price * count) + (global_mean * self.smoothing)) / (count + self.smoothing)
        self.category_mean_price = self.category_mean_price.reset_index()
        self.category_mean_price.columns = ['OutletID', 'Type', 'TypeMeanPrice']
        return self

    def transform(self, X):
        X = X.copy()
        # Merge the mean price with the original dataframe
        X = pd.merge(X, self.category_mean_price, on=['OutletID', 'Type'], how='left')
        X['TypeMeanPrice'] = np.log1p(X['TypeMeanPrice'])
        return X[['TypeMeanPrice']]

```

```
class VisibilityZerosImputerZerosImputer(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.item_Visibility_mean = defaultdict(lambda: None)  # Stores mean of Visibility per ItemID
        self.global_mean = None  # Fallback global mean for Visibility

    def fit(self, X, y=None):
        X = X.copy()

        # Ensure the required columns exist
        if 'ItemID' not in X.columns or 'Visibility' not in X.columns:
            raise ValueError("Both 'ItemID' and 'Visibility' columns must be present in the dataset.")

        # Ensure no None values in ItemID
        if X['ItemID'].isnull().any():
            raise ValueError("ItemID column contains None values.")

        # Calculate mean Visibility for each ItemID
        item_Visibility_mean = X.groupby('ItemID')['Visibility'].mean()
        self.item_Visibility_mean.update(item_Visibility_mean.to_dict())

        # Calculate global mean for the Visibility column
        if X['Visibility'].notnull().any():
            self.global_mean = X['Visibility'].mean()
        else:
            raise ValueError("Visibility column contains only NaN values.")

        return self

```

```
    def transform(self, X):
        X = X.copy()

        # Ensure the required columns exist
        if 'ItemID' not in X.columns or 'Visibility' not in X.columns:
            raise ValueError("Both 'ItemID' and 'Visibility' columns must be present in the dataset.")

        # Ensure no None values in ItemID
        if X['ItemID'].isnull().any():
            raise ValueError("ItemID column contains None values.")

        # Safely impute zero Visibilitys based on ItemID or global mean
        X['Visibility'] = X.apply(
            lambda row: self.item_Visibility_mean.get(row['ItemID'], self.global_mean)
            if row['Visibility'] == 0
            else row['Visibility'],
            axis=1
        )

        X['Visibility'] = X['Visibility'].replace(0, self.global_mean)
        X['Visibility'] = np.sqrt(X['Visibility'])

        return X[['Visibility']]

```

```
class CustomFeatureTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, column_names):
        self.column_names = column_names
        self.scaler_price_per_unit_weight = StandardScaler()
        self.scaler_X2 = StandardScaler()
        self.scaler_pricing_strategy = StandardScaler()

    def fit(self, X, y=None):
        # Convert to DataFrame with updated column names
        X = pd.DataFrame(X, columns=self.column_names)
        # Create new features
        X['price_per_unit_weight'] = X['TypeMeanPrice_0'] / (X['X2'] + 0.0001)
        X['pricing_strategy'] = X['MRP'] - (X['TypeMeanPrice_0'] * (X['Outlet_TypeOrdinal'] + 1) * (X['Location_Type'] + 1))

        # Fit the scalers on the training data
        self.scaler_price_per_unit_weight.fit(X[['price_per_unit_weight']])
        self.scaler_X2.fit(X[['X2']])
        self.scaler_pricing_strategy.fit(X[['pricing_strategy']])

        return self

    def transform(self, X):
        # Convert to DataFrame with updated column names
        X = pd.DataFrame(X, columns=self.column_names)

        # Create new features
        X['BigMac_index2'] = (X['Visibility'] + 0.0001) * (X['Outlet_TypeOrdinal'] + 1) * (X['TypeMeanPrice_0'] + 3.24995) * (X['Location_Type'] + 1)
        X['price_per_unit_weight'] = X['TypeMeanPrice_0'] / (X['X2'] + 0.0001)
        X['BigMac_index2'] = np.sqrt(X['BigMac_index2'])
        X['pricing_strategy'] = X['MRP'] - (X['TypeMeanPrice_0'] * (X['Outlet_TypeOrdinal'] + 1) * (X['Location_Type'] + 1))
        X['RegionalPotential'] = X['Location_Type'] * (X['PastYearsCont'] + 1)
```

```

        # Standard scale the new features
        X['price_per_unit_weight'] = self.scaler_price_per_unit_weight.transform(X[['price_per_unit_weight']])
        X['X2'] = self.scaler_X2.transform(X[['X2']])
        X['pricing_strategy'] = self.scaler_pricing_strategy.transform(X[['pricing_strategy']])

        # Drop unwanted columns
        drop_columns = [
            "Category_DR", "Category_FD", "Outlet_Type_Supermarket Type2",'FatContent',
            "Outlet_Type_Supermarket Type1", "BigMac_index3",'Size',
            "Visibility", "BigMac_index", "Outlet_Type_Grocery Store", "Outlet_Type_Supermarket Type3"
        ]
        X.drop(columns=[col for col in drop_columns if col in X.columns], inplace=True)

        return X

```

```
class KMeansTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, num_clusters=5, random_state=42):
        self.num_clusters = num_clusters
        self.random_state = random_state


    def fit(self, X, y=None):
        #X = X.drop("price_per_unit_weight", axis=1)
        # Fit KMeans using the original features
        self.feature_names_in_ = X.columns  # Store original column names
        self.kmeans = KMeans(n_clusters=self.num_clusters, random_state=self.random_state)
        self.kmeans.fit(X)
        return self

    def transform(self, X):
        #X = X.drop("price_per_unit_weight", axis=1)
        # Ensure the input columns match those seen during fit
        if list(X.columns) != list(self.feature_names_in_):
            raise ValueError(
                "The feature names should match those that were passed during fit.\n"
                f"Feature names unseen at fit time: {set(X.columns) - set(self.feature_names_in_)}\n"
                f"Feature names seen at fit time but not in transform: {set(self.feature_names_in_) - set(X.columns)}"
            )

        # Compute cluster labels
        cluster_labels = self.kmeans.predict(X)

        # Compute distances to centroids
        centroid_distances = self.kmeans.transform(X)

        # Create a new DataFrame with cluster labels and centroid distances
        new_features = pd.DataFrame(
            centroid_distances,
            columns=[f"Centroid_{i}" for i in range(centroid_distances.shape[1])],
            index=X.index
        )
        new_features['Cluster'] = cluster_labels

        # Concatenate the original DataFrame with the new features
        X_transformed = pd.concat([X.reset_index(drop=True), new_features.reset_index(drop=True)], axis=1)
```

```
        return X_transformed
```

```
all_features = ['EstablishmentYear', 'MRP',"Outlet_Type",'MRP_cluster','Snack Foods', 'Frozen Foods', 'Fruits and Vegetables', 'Canned','X2','Visibility']
target_encoder_cols = ['Type']
numerical_cols = ['EstablishmentYear', 'MRP']
VisibilityZeros_cols = ['ItemID','Visibility']
X2_imputer_cols = ['X2','ItemID']
ordinal_cols = [ "Outlet_Type",'MRP_cluster']
TypeMeanPriceTransformer_col = ['OutletID' ,'Type']
one_hot_columns = ['Type']  # only one-hot encoding
ordinal_categories = [  # FatContent
    ['Grocery Store', 'Supermarket Type1', 'Supermarket Type2', 'Supermarket Type3'],
    ['very low','low','high','very high']

   # Size
]

# Define numerical pipeline
numerical_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent"))
    #,("scaler", StandardScaler())
])

# size_imputer_pipeline = Pipeline([
#     ("custom_size_imputer", SizeImputer()),
#     ("imputer", SimpleImputer(strategy="most_frequent")),
#     ("ordinal", OrdinalEncoder(categories=[['Small', 'Medium', 'High']]))
# ])

type_mean_price_pipeline = Pipeline([
    ("type_mean_price_transformer", TypeMeanPriceTransformer(smoothing=0.5)),
    ("imputer", SimpleImputer(strategy="mean"))
])

# Define one-hot encoding pipeline
ordinal_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("ordinal", OrdinalEncoder(categories=ordinal_categories))
])

# Define one-hot encoding pipeline for onehot-transform columns
onehot_transform_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(categories=[['Snack Foods', 'Frozen Foods', 'Fruits and Vegetables', 'Canned']], handle_unknown="ignore"))
])
# Define target encoding pipeline
target_encoding_pipeline = Pipeline([
    ("target_encoder", TargetEncoder(cols=target_encoder_cols , smoothing=0.5) )
])

# Combine all pipelines into a ColumnTransformer
preprocessor = ColumnTransformer([
    ("num", numerical_pipeline, numerical_cols),
   # ("type_mean_price", type_mean_price_pipeline, TypeMeanPriceTransformer_col),
    #("target_encoder", target_encoding_pipeline, target_encoder_cols),
    ("ordinal", ordinal_pipeline, ordinal_cols),
    ("onehot_", onehot_transform_pipeline, one_hot_columns),
```

```
    ("X2_imputer", X2NaNsImputer(), X2_imputer_cols),
    ("VisibilityZeros", VisibilityZerosImputerZerosImputer(), VisibilityZeros_cols)
])

# Define the final pipeline
# Create the extended pipeline
final_pipeline = Pipeline([
    ("preprocessor", preprocessor)
    #("custom_features", CustomFeatureTransformer(column_names=all_features )),
    #("kmeans", KMeansTransformer(num_clusters=5, random_state=42))
])



# Fit and transform the data

# Get the feature names after transformation

```

```
import optuna
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

def objective_xgb(trial):
    param = {
        'n_estimators': trial.suggest_int('n_estimators', 50, 1000),
        'learning_rate': trial.suggest_float('learning_rate', 0.001, 0.3),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'subsample': trial.suggest_float('subsample', 0.4, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.4, 1.0),
        'reg_alpha': trial.suggest_float('reg_alpha', 0, 10),
        'reg_lambda': trial.suggest_float('reg_lambda', 0, 10),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 10),
        'gamma': trial.suggest_float('gamma', 0, 5),
        'max_bin': trial.suggest_int('max_bin', 128, 512),
        'objective': 'reg:absoluteerror',
        'eval_metric': 'mae',
        'random_state': 42,
        'enable_categorical': True
    }

    model = XGBRegressor(**param)
    scores = cross_val_score(model, X_trans, Y, cv=5, scoring='neg_mean_absolute_error')
    mae = -scores.mean()
    return mae

study_xgb = optuna.create_study(direction='minimize')
study_xgb.optimize(objective_xgb, n_trials=70)

print("XGBoost Best Parameters:", study_xgb.best_params)
print("XGBoost Best MAE:", study_xgb.best_value)

```

```
best_xgb_model.fit(X_train, Y_train)

# Calculate feature importances
```

```python
mdi_importances = pd.Series(best_xgb_model.feature_importances_, index=X_train.columns)
tree_importance_sorted_idx = np.argsort(best_xgb_model.feature_importances_)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 8))

# Plot Gini importance
mdi_importances.sort_values().plot.barh(ax=ax1)
ax1.set_xlabel("Gini importance")

# Plot permutation importance
plot_permutation_importance(best_xgb_model, X_train, Y_train, ax2)
ax2.set_xlabel("Decrease in accuracy score")

fig.suptitle("Impurity-based vs. permutation importances on multicollinear features (train set)")
fig.tight_layout()
plt.show()
```

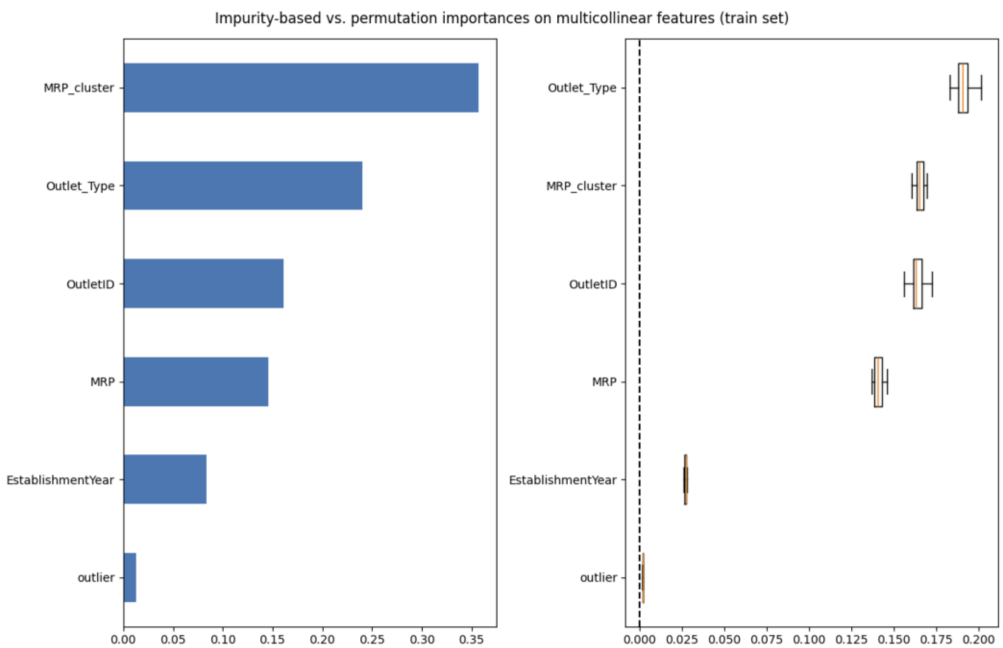
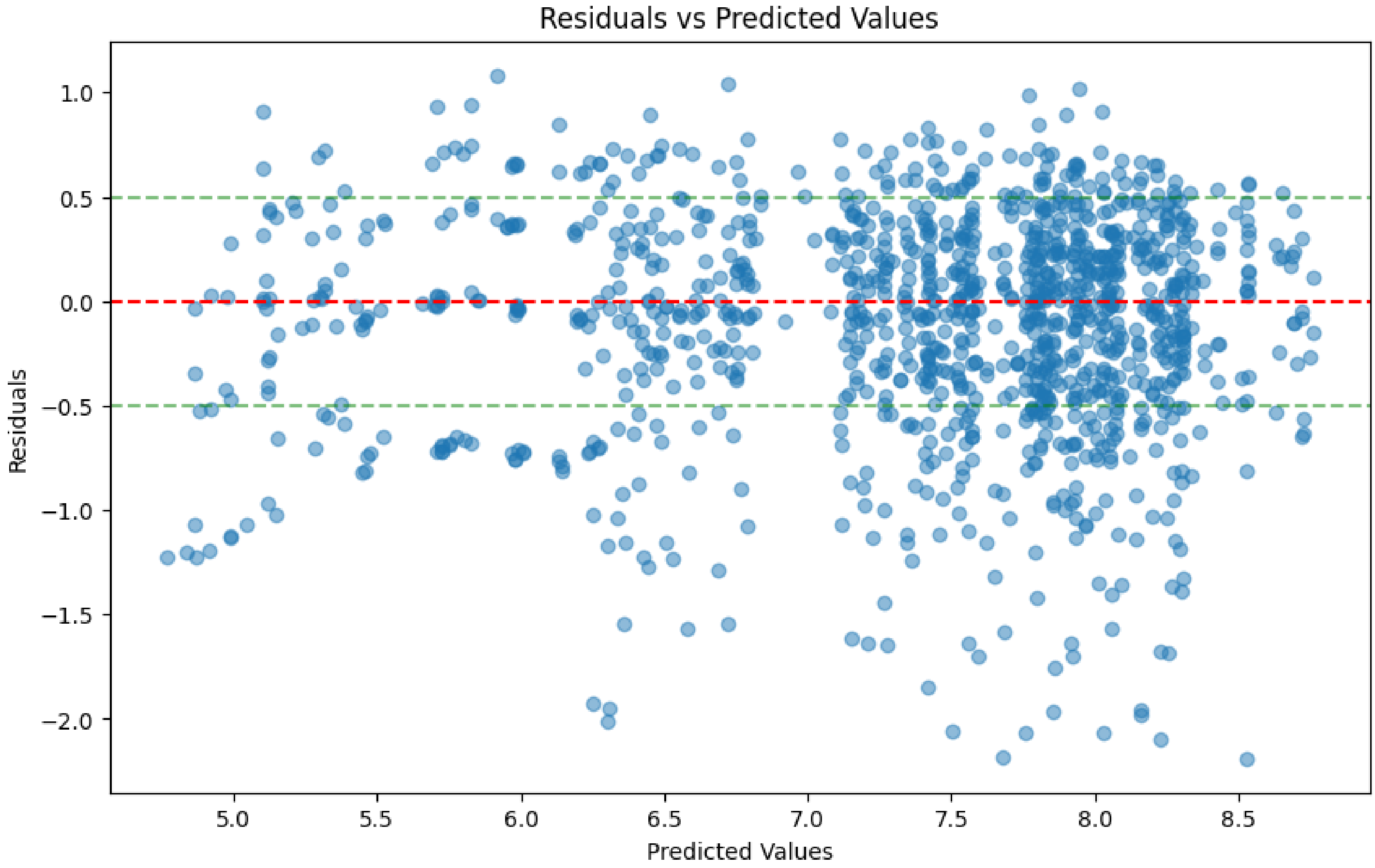


Figure 11 Impurity-based vs. permutation importances on multicollinear features

Residual Model:



Insights About Model Performance

Overall Bias:

- The residuals are centered around 0, meaning the model generally produces unbiased predictions.
- There is no clear indication of systematic over- or under-prediction across the full range of predicted sales prices.
- This is a positive indicator of the model's calibration.

Heteroscedasticity:

- The spread of residuals appears to increase for lower and higher predicted sales prices (e.g., below 6 and above 8). This suggests heteroscedasticity, where the model's prediction error varies with the magnitude of the predicted sales price.

  - For low-price products, the model shows higher residual variability, indicating the predictions for these products are less accurate.
  - Similarly, high-price products (above ~8) also exhibit larger prediction errors.

Outliers:

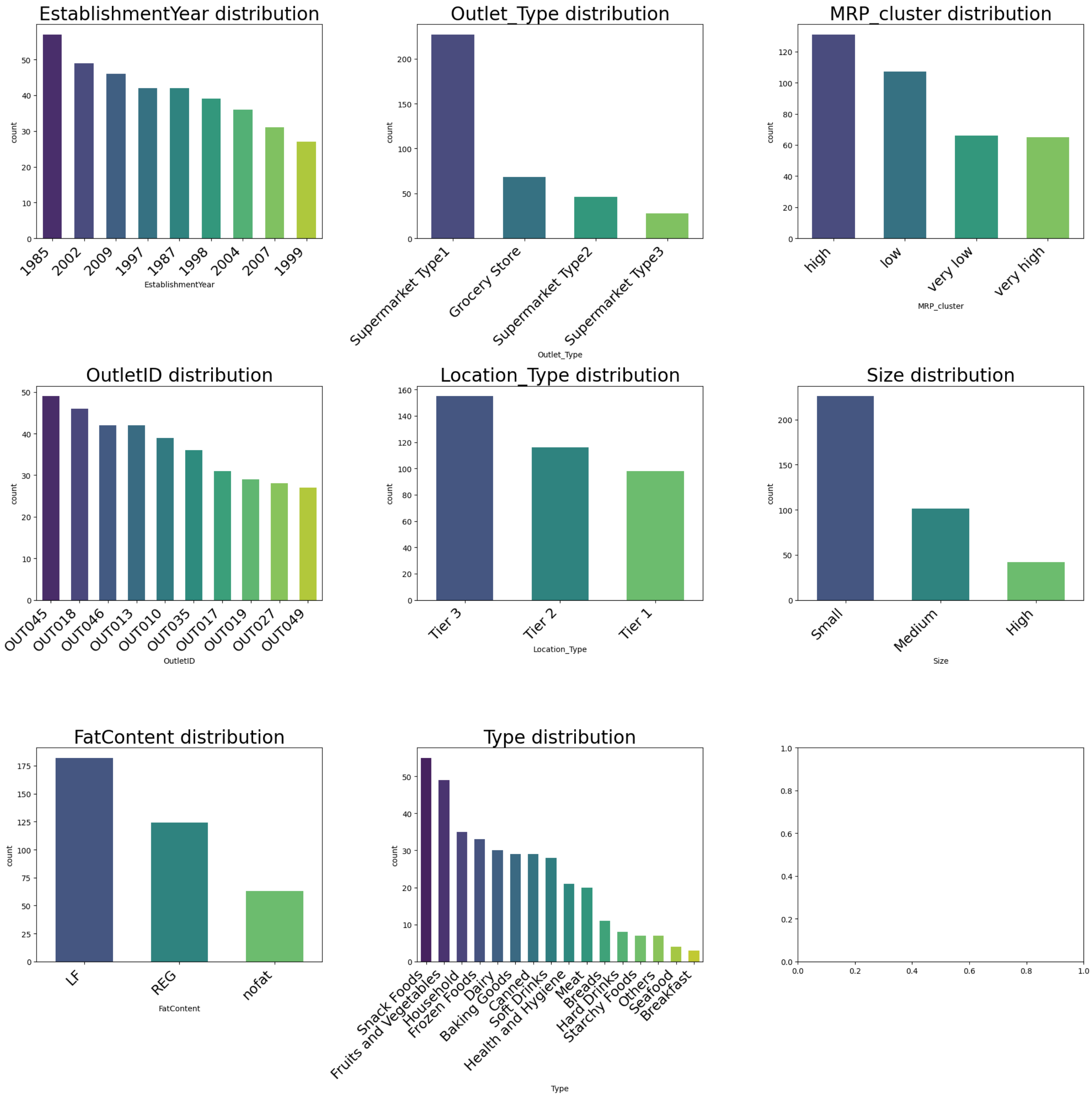
- There are a few outliers with large residuals (e.g., <-1.5 or >1.0). These points correspond to extreme cases where the model predictions deviate significantly from the actual sales price. These outliers could result from:
  - Noise in the data (e.g., data entry errors or rare products).
  - Insufficient representation of these cases in the training data.

Prediction Accuracy Across Ranges:

- The residuals for mid-range predicted prices (6.5–7.5) are relatively well-behaved, with smaller spreads.
- This suggests that the model performs better for mid-priced products compared to low- or high-priced ones.

Residuals > abs(0.5):

High residual Dataframe:




```
X_trans['outlier'] =( X_trans['Snack Foods'] + X_trans['Frozen Foods'] + X_trans['Fruits and Vegetables'] +  X_trans['Canned']) * (X_trans['X2'] + X_trans['Visibility'])
```