Predictive Modeling for Demand Forecasting and Maintenance in Grocery Stores

U00871097

Introduction:

This report uses previous sales data to create a model for a grocery store's sales forecasting. The number of units sold for a specific week is predicted using the Random Forest Regressor method from the Scikit-Learn module. A dataset including weekday and other relevant feature information is used to train the model. This data-driven strategy offers insights into potential sales, improving inventory control and decision-making.

Dataset:

The "grocery\_store\_data.csv" training dataset includes historical sales information for the grocery store, such as the average weekly sales volume and other characteristics. In order to make the dataset easier for the model to handle, duplicates and missing data are removed, and the week column is changed to a Unix timestamp. By forecasting the number of units sold in accordance with the model we will train using the train data, "grocerytestdata.csv" is used to test the model.

Preprocessing and Model Development:

Data preprocessing is an essential stage in any machine learning project since it gets the raw data ready for modeling and boosts the algorithm's overall performance. We carried out the following data pretreatment operations in the project to estimate grocery shop sales:

Loading the dataset:

A pandas DataFrame was created once the "grocery\_store\_data.csv" file was loaded. This file includes elements like "week" and other factors pertaining to the items being sold, along with historical sales data for the grocery store.

Converting the 'week' column to a Unix timestamp:Date information is presented in the 'week' column using the 'dd/mm/yy' format. We changed the 'week' field to a Unix timestamp to simplify numerical processing and model training. Unix timestamps are better suited for numerical analysis because they show the amount of seconds since January 1, 1970.

Removing Duplicate entries: Duplicate entries can result in biased model training and overfitting. Using the 'drop\_duplicates()' function in pandas, we eliminated any duplicate rows from the dataset to assure the accuracy of the data.

Removing missing data: Missing data might have a detrimental effect on the model's performance. We removed any rows with missing values using the 'dropna()' method in pandas to manage missing data. Thus, we can be certain that our dataset is comprehensive and prepared for model training.

Defining features and target variable: We divided the dataset into the features (X) and the target variable (y) before training the model. The 'units\_sold' target variable gives the total number of units sold for each item in the dataset. 'units\_sold' and 'week' are excluded from the features, which include all other columns in the dataset. The model will be trained using these features, and predictions for the test dataset will be made using it.

Splitting the dataset into training and testing sets: We divided the dataset into training and testing sets to assess the model's performance. For training, we used 80% of the data, and for testing, 20%. This division enables us to train the model on a bigger subset of the data and assess its performance on a different, unobserved subset. The 'train\_test\_split()' function from the sklearn library was used to carry out this split.

These data pretreatment procedures allowed us to confirm the accuracy of the input data while also preparing the sales data from grocery stores for modeling. To create machine learning models that are accurate and dependable, proper data preprocessing is necessary.

I tried fitting the data in linear regression and the r2 scores were too low. Linear regression would be better if I had more variables.

# Fit a linear regression model

lr = LinearRegression()

lr.fit(X\_train, y\_train)

y\_pred\_lr = lr.predict(X\_test)

# Calculate the R2 score for the linear regression model

r2\_lr = r2\_score(y\_test, y\_pred\_lr)

# Fit the polynomial regression model on the training data

poly = PolynomialFeatures(degree=2)

X\_poly\_train = poly.fit\_transform(X\_train)

X\_poly\_test = poly.transform(X\_test)

lr\_poly = LinearRegression()

lr\_poly.fit(X\_poly\_train, y\_train)

y\_pred\_poly = lr\_poly.predict(X\_poly\_test)

# Fit a polynomial regression model with degree=2

pr = LinearRegression()

pr.fit(X\_poly\_train, y\_train)

X\_poly\_test = poly.transform(X\_test)

y\_pred\_pr = pr.predict(X\_poly\_test)

My Output is as follows:

Linear Regression R2 score: 0.26488469385783775

Polynomial Regression R2 score: 0.349293472955564

Polynomial Regression MSE: 2129.4046135143753

Polynomial Regression RMSE: 46.145472297012795

This is not at all efficient for my data model, so I thought to use random forest regressor.

A Random Forest Regressor is instantiated and trained on 80% of the dataset, with the remaining 20% reserved for testing.

Based on the training data, we then trained a Random Forest Regressor. An ensemble machine learning approach called Random Forest is frequently employed for regression issues. To produce a final forecast, the algorithm builds many decision trees and then combines their predictions.

The model's parameters were adjusted using a grid search to further enhance the performance of the Random Forest Regressor. In order to do the grid search, a range of values for the parameters (n\_estimators, max\_depth, and max\_leaf\_nodes) had to be established. The model had to be trained using each set of parameters, and its performance was assessed using the R2 score and RMSE.

I prefer not to use scaling because I tried using scaling but the efficiencies have gone down due to that.

Then, the parameters with the highest R2 score and lowest RMSE were chosen as the best ones. The collection of parameters that provided the best harmony between accuracy and goodness of fit was the outcome of the grid search.

The output before running the parameter tuning is as follows:

Random Forest Regressor:

R2 score: 0.770

MSE: 752.405

RMSE: 27.430

The final model was trained using the grid search results, and it was then used to test data to make predictions. The analysis also included the results of the grid search and the final model parameters.

# Defining a range of values for the parameters to search

n\_estimators\_range = [10, 100, 150]

max\_depth\_range = [10, 30, 50, None]

max\_leaf\_nodes\_range = [10, 30, 50, None]

# Initializing the best parameters and the best scores

best\_params = None

best\_r2 = float('-inf')

best\_rmse = float('inf')

# Looping through the parameter grid

for n\_estimators in n\_estimators\_range:

    for max\_depth in max\_depth\_range:

        for max\_leaf\_nodes in max\_leaf\_nodes\_range:

            # Instantiation of the Random Forest Regressor with the current parameters

            rf = RandomForestRegressor(n\_estimators=n\_estimators, max\_depth=max\_depth, max\_leaf\_nodes=max\_leaf\_nodes, random\_state=42)

            # Training the model

            rf.fit(X\_train, y\_train)

            # Prediction and evaluation of the model with R2 and RMSE

            y\_pred = rf.predict(X\_test)

            r2 = r2\_score(y\_test, y\_pred)

            mse = mean\_squared\_error(y\_test, y\_pred)

            rmse = np.sqrt(mse)

            # Updating the best parameters and the best scores if both conditions are met

            if r2 > best\_r2 and rmse < best\_rmse:

                best\_params = {'n\_estimators': n\_estimators, 'max\_depth': max\_depth, 'max\_leaf\_nodes': max\_leaf\_nodes}

                best\_r2 = r2

                best\_rmse = rmse

# Print the best parameters and the best scores

print(f"Best parameters: {best\_params}")

print(f"Best R2 score: {best\_r2:.3f}")

print(f"Best RMSE: {best\_rmse:.3f}")

Output:

Best parameters: {'n\_estimators': 150, 'max\_depth': 30, 'max\_leaf\_nodes': None}

Best R2 score: 0.772

Best RMSE: 27.306

After assessment of best parameters I just used the best parameters in my program and skimmed away the rest of code for grid search. My final code is as follows:

# Loading dataset into a pandas dataframe

df = pd.read\_csv("grocery\_store\_data.csv")

# Converting the 'week' column to a Unix timestamp

df['week\_unix'] = pd.to\_datetime(df['week'], format='%d/%m/%y')

df['week\_unix'] = df['week\_unix'].astype(np.int64) // 10\*\*9

# Removing the duplicates

df.drop\_duplicates(inplace=True)

# Removing the missing data

df.dropna(inplace=True)

# Defining the features and target variable

X = df.drop(["units\_sold", "week"], axis=1)

y = df["units\_sold"]

# Splitting the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Instantiation of the Random Forest Regressor

rf = RandomForestRegressor(n\_estimators=150, max\_depth=30, max\_leaf\_nodes=None, random\_state=42)

# Training the model

rf.fit(X\_train, y\_train)

# Prediction and evaluation of the model with r2, mean squared error, and root mean squared error

y\_pred = rf.predict(X\_test)

r2 = r2\_score(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

rmse = np.sqrt(mse)

# Print the metrics

print("Random Forest Regressor:")

print(f"R2 score: {r2:.3f}")

print(f"MSE: {mse:.3f}")

print(f"RMSE: {rmse:.3f}")

We assessed the model's performance using the R2 score, Mean Squared Error (MSE), and Root Mean Squared Error (RMSE) after training it on the testing data. A score of 1 indicates a perfect fit, while a score of 0 indicates a poor fit, and the R2 score is a measurement of how well the model matches the data. Lower values of the MSE and RMSE indicate a better match as they quantify the difference between the true values and the predicted values.

# Prediction and evaluation of the model with r2, mean squared error, and root mean squared error

y\_pred = rf.predict(X\_test)

r2 = r2\_score(y\_test, y\_pred)

mse = mean\_squared\_error(y\_test, y\_pred)

rmse = np.sqrt(mse)

# Print the metrics

print("Random Forest Regressor:")

print(f"R2 score: {r2:.3f}")

print(f"MSE: {mse:.3f}")

print(f"RMSE: {rmse:.3f}")

The model is evaluated using three metrics: R2 score, Mean Squared Error (MSE), and Root Mean Squared Error (RMSE). The results are as follows:

Random Forest Regressor:

R2 score: 0.772

MSE: 745.522

RMSE: 27.304

These results indicate that the model can explain 77.2% of the variance in the data, and the RMSE value of 27.304 suggests that the model's predictions have an average deviation of around 41 units from the actual values.

Sales Prediction:

A different test dataset called "grocerytestdata.csv" is used to estimate future sales. This dataset's week field has also been converted to a Unix timestamp for forecasting purposes. The model is applied to forecast the number of units sold for each item in the test dataset. The projected values are now stored in a new column called "predicted\_units\_sold" in the test dataset. The original week column is retained in the final output, which also includes all original columns and the projected number of units sold.

# Loading the separate test dataset

test\_df = pd.read\_csv("grocerytestdata.csv")

# Storing the original week values

original\_week = test\_df['week'].copy()

# Converting the 'week' column to a Unix timestamp for prediction purposes

test\_df['week\_unix'] = pd.to\_datetime(test\_df['week'], format='%d/%m/%y')

test\_df['week\_unix'] = test\_df['week\_unix'].astype(np.int64) // 10\*\*9

# Prediction for the separate test dataset

test\_pred = rf.predict(test\_df.drop("week", axis=1))

# Adding the predicted units\_sold to the test dataset

test\_df['predicted\_units\_sold'] = test\_pred

# Dropping the 'week\_unix' column from the test dataset

test\_df.drop("week\_unix", axis=1, inplace=True)

# Saving the updated test dataset with predicted units\_sold back to the original file

test\_df.to\_csv("grocerytestdata.csv", index=False)

# Print the first few rows of the test dataset with predicted units\_sold and other columns

print(test\_df.head())

Conclusion:

Future grocery store sales can be accurately predicted using the Random Forest Regressor-based sales forecasting model more accurately than linear regression with few variables available to me. I improved the r2 value by performing the parameter tuning using grid search. The model performs well, showing its potential utility in inventory management and decision-making with an R2 score of 0.772. The model can be made to perform better by adding additional features or using more sophisticated methods. My final output after predicting the units\_sold in test data as 'predicted\_units\_sold’ is as follows:

Random Forest Regressor:

R2 score: 0.772

MSE: 745.522

RMSE: 27.304

week record\_ID store\_id sku\_id total\_price base\_price \

0 16/07/13 212645 8091 216418 108.3000 108.3000

1 16/07/13 212646 8091 216419 109.0125 109.0125

2 16/07/13 212647 8091 216425 133.9500 133.9500

3 16/07/13 212648 8091 216233 133.9500 133.9500

4 16/07/13 212649 8091 217390 176.7000 176.7000

is\_featured\_sku is\_display\_sku week\_unix predicted\_units\_sold

0 0 0 1373932800 12.386667

1 0 0 1373932800 22.680000

2 0 0 1373932800 38.226667

3 0 0 1373932800 34.206667

4 0 0 1373932800 48.040000