Stores Sales Forecasting

Author(s):
Hamza Naser
Diaa Alqadi
Dana Ghazal

Sep 20th, 2023

Table of Contents

Chapter 1
Introduction

In today's dynamic and competitive retail landscape, the ability to accurately predict future sales is paramount for business success. Store owners and managers need reliable tools to anticipate customer demand, optimize inventory management, and make data-driven decisions that can ultimately boost profitability. One such tool that has gained significant traction in recent years is machine learning (ML), specifically the use of advanced algorithms to predict weekly sales with a high degree of accuracy.

Store Sales Forecasting using machine learning algorithms is a cutting-edge approach that leverages historical sales data, and external factors to provide forecasts that are not only precise but also adaptable to changing market conditions. By harnessing the power of machine learning, businesses can gain a competitive edge by aligning their supply chain, staffing, and marketing strategies with anticipated demand, resulting in improved customer satisfaction and increased revenue.

In this project, we will conduct an analysis of Walmart sales data for 45 stores over the time period from February 5, 2010, to October 26, 2012. The dataset includes various features such as store information, department details, economic indicators such as the Consumer Price Index, unemployment rate, weather information, promotional markdowns, fuel prices, and more. Our primary objective is to employ machine learning algorithms to forecast weekly sales for each store, leveraging this extensive set of features.

1.1. Problem Definition

In the face of underperforming sales and the pressing need for growth, this project centers on devising a comprehensive strategy to augment store sales and attract a higher volume of customers within the next three months. The store finds itself operating within a fiercely competitive retail landscape, which necessitates innovative approaches to meet desired sales targets and expand its market presence. Over this short-term horizon, the objective is to realize a 5% surge in customer numbers.

Our success benchmarks revolve around two primary goals. The first is to increase weekly sales by 10%, this will be achieved through a strategic expansion plan, entailing the establishment of new store branches across diverse regions. These new locations are expected to contribute significantly to the overall sales growth. The second goal is to enhance customer engagement and experience, as customer satisfaction and loyalty are pivotal. We aim to achieve this through enhanced customer engagement strategies and an enriched shopping experience, ensuring customers return and recommend the store to others.

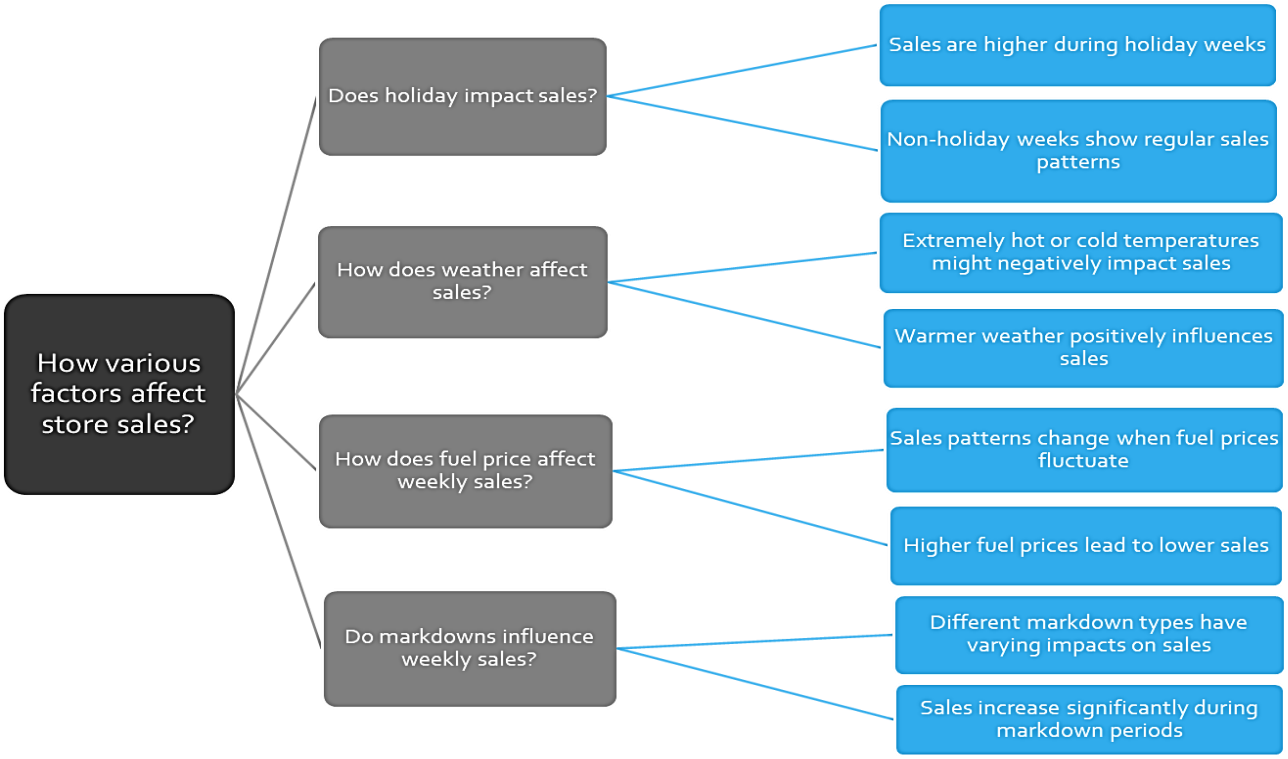
But several constraints shape the parameters of this project. The foremost constraint revolves around financial resources, where budgetary limitations prevail. Limited funds must be allocated wisely, earmarked for the dual purpose of opening new store branches and implementing strategies geared towards amplifying sales. Another pivotal constraint pertains to legal and ethical compliance. All marketing and sales activities must adhere strictly to legal and ethical standards, ensuring that the strategies employed align with industry regulations and uphold the store's reputation.

The project will concentrate on the 45 existing stores, with particular emphasis on regions that have demonstrated the highest sales potential. By focusing efforts on these key areas and considering the outlined constraints and objectives, we aim to revitalize the store's performance and foster sustainable growth over the next three months.

1.2. Problem Structuring

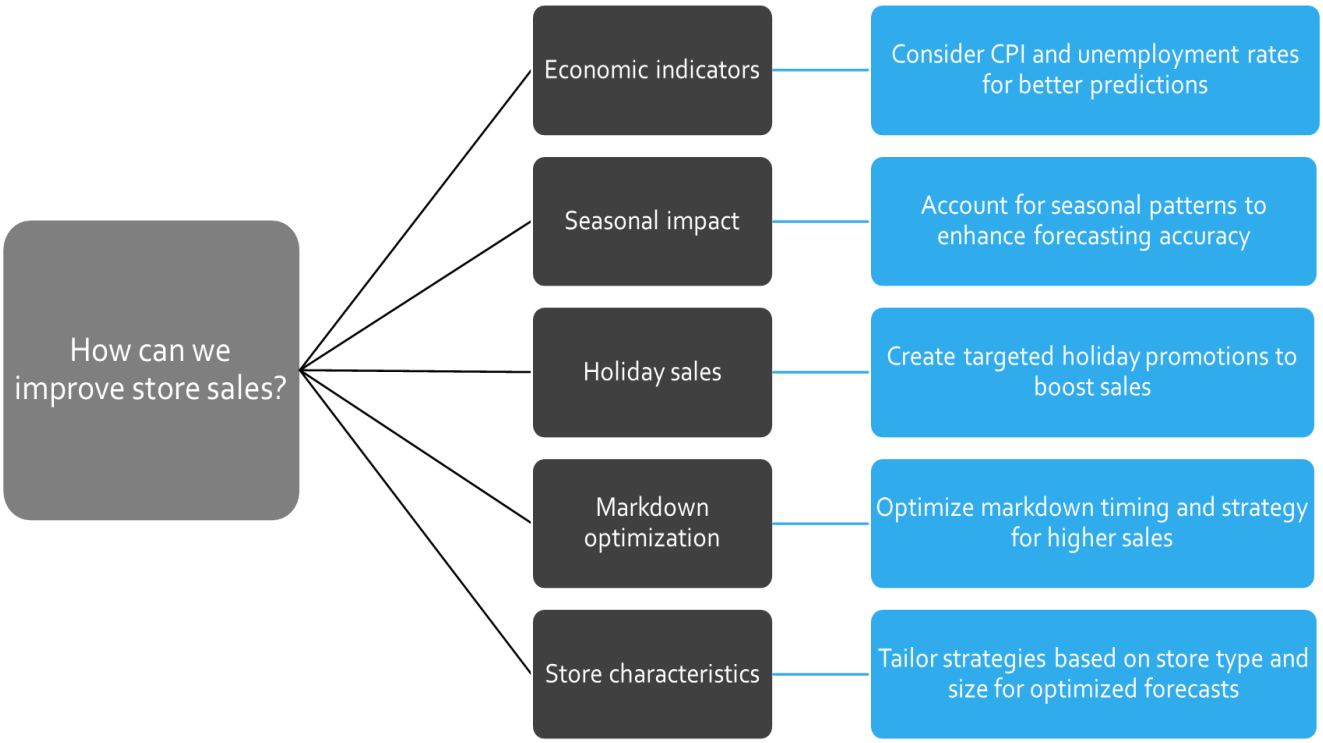
- **Hypothesis Tree**

In our pursuit of unraveling the intricate dynamics of store sales, we have constructed a comprehensive Hypothesis Tree that systematically explores the multifaceted factors influencing sales performance. This structured framework serves as our guiding compass, enabling us to probe, analyze, and validate various hypotheses pertaining to the intricate interplay of variables.



- **Deductive Tree**

Within the framework of our Deductive Tree, we embark on a structured exploration of strategies aimed at elevating store sales to new heights. This methodical approach involves dissecting the multifaceted challenge of improving sales through a deductive reasoning process.



Chapter 2
Progress & Implementation

In this chapter, we detail our preparations for implementation, including dataset study and preparation. We examine our code and experiments to ensure data quality. The chapter's core focuses on training and testing our machine learning model.

2.1. Datasets

The dataset to this project comprises three tables, each in the widely used .CSV file format. These tables collectively furnish comprehensive insights into the sales dynamics of 45 Walmart stores, each strategically positioned in diverse geographical regions.

1. **Stores Dataset:** This table offers essential general information pertaining to each of the 45 stores under consideration. With a structured format of 45 rows and 3 columns, it provides a foundational understanding of each store's unique characteristics.

Column	Description	Data Type
Store	Stores numbers from 1 to 45	Categorical
Type	Store type has been provided, there are 3 types A, B, and C	Categorical
Size	Stores size	Numerical

2. **Train Dataset:** This dataset contains the target variable, which is weekly sales data. It accurately records the sales figures for each store and department. It comprises an impressive 421,570 rows and 5 columns, offering a wealth of historical sales data essential for training and modeling our predictive algorithms.

Column	Description	Data Type
Store	The store number	Categorical
Dept	The department number	Categorical
Date	Day of the week	Categorical
Weekly_Sales	Sales for the given store in Dollars	Numerical
IsHoliday	Whether the week is a special holiday week	Categorical

3. **Features Dataset:** Comprising 8,190 rows and 12 columns, this table encapsulates an array of store-specific data. It is thoughtfully categorized into two main sections: Regional Information, encompassing variables like temperature and unemployment rates, and Promotional Information, elucidating various mark down options employed during different business weeks.

Column	Description	Data Type
Store	The store number	Categorical
Date	Day of the week	Categorical
Temperature	Average temperature in the region in Fahrenheit	Numerical
Fuel_Price	Cost of fuel in the region in Dollars	Numerical
MarkDown1	Anonymized data related to promotional markdowns that Walmart is running	Numerical
MarkDown2	Anonymized data related to promotional markdowns that Walmart is running	Numerical
MarkDown3	Anonymized data related to promotional markdowns that Walmart is running	Numerical
MarkDown4	Anonymized data related to promotional markdowns that Walmart is running	Numerical
MarkDown5	Anonymized data related to promotional markdowns that Walmart is running	Numerical
CPI	The consumer price index	Numerical
Unemployment	The unemployment rate	Numerical
IsHoliday	Whether the week is a special holiday week	Categorical

2.2. Datasets Preprocessing

Data preprocessing plays a crucial role in the development of machine learning models. We will outline the steps taken to prepare our dataset using the Python programming language and its various libraries. Initially, we downloaded the datasets and subsequently embarked on the preprocessing journey.

- **Checking and Handling the NULL Values**

We checked for any missing values, and it turns out that the Features table contains null values in the five MarkDown columns (1-5), CPI column, and Unemployment column as shown in the figure below. Therefore, we handled missing values in the markdown columns by replacing them with zeros, and we handled the nulls in the CPI and unemployment columns by creating a linear regression model to calculate and fill the nulls based on Date and Store number.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8190 entries, 0 to 8189
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Store         8190 non-null   int64
 1   Date          8190 non-null   object
 2   Temperature   8190 non-null   float64
 3   Fuel_Price    8190 non-null   float64
 4   MarkDown1     4032 non-null   float64
 5   MarkDown2     2921 non-null   float64
 6   MarkDown3     3613 non-null   float64
 7   MarkDown4     3464 non-null   float64
 8   MarkDown5     4050 non-null   float64
 9   CPI           7605 non-null   float64
 10  Unemployment  7605 non-null   float64
 11  IsHoliday     8190 non-null   bool
dtypes: bool(1), float64(9), int64(1), object(1)
memory usage: 712.0+ KB
```

- **Checking for Negative Values**

Subsequently, upon careful examination, we identified the presence of negative values in Markdown columns (1-3) and 5 of the Features table, as well as a negative value of 1285 in the Weekly Sales column in the Train table. As illustrated in the figures below. But we made a decision to retain these negative values within our dataset. The rationale behind this choice was twofold: firstly, to preserve the integrity and completeness of our data to ensure that our machine learning model would perform, and secondly, to acknowledge the possibility that these negative values may indeed represent financial losses incurred during specific periods.

	MarkDown1	MarkDown2	MarkDown3	MarkDown4	MarkDown5
min	-2781.45	-265.76	-179.26	0.22	-185.17

	Weekly_Sales
min	-4988.94

- **Check and Correct the Data Type**

We noticed that the data type of some columns is wrong, so we changed the data type as shown in the table below, Store to object, Dept to object, and Date to datetime.

Column	Type	Correct Type
Store	int64	object
Dept	int64	object
Date	object	datetime64

- **Checking for Outliers**

We found that we have outlier's values in six columns in the datasets in the Weekly Sales column and the five MarkDown columns (1-5), and the following figure shows the number of outliers in each columns and shows that the Weekly Sales column has the highest number of outliers. But since we are trying to forecast weekly sales it's not a wise move to drop outliers since this will affect the model's performance and the ability to predict the sales for out-of-sample data, and the same goes for markdown. So we decided not to remove them and keep them in the dataset.

```
The number of outliers of the column Weekly_Sales is 35521
The number of outliers of the column MarkDown1 is 237
The number of outliers of the column MarkDown2 is 436
The number of outliers of the column MarkDown3 is 480
The number of outliers of the column MarkDown4 is 337
The number of outliers of the column MarkDown5 is 212
```

- **Convert the Unit**

To enhance interpretability, we converted the values in the Temperature column from Fahrenheit to Celsius.

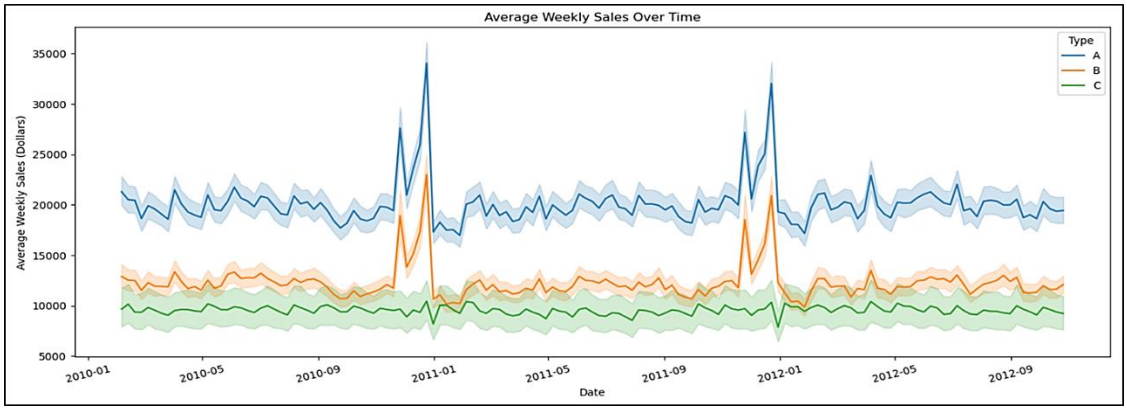
- **Merge Datasets**

We have merged the three distinct tables, namely the Features table, Stores table, and Train table, into a consolidated and comprehensive master dataset as shown in the figure below. This master dataset will contain a harmonized representation of store-specific information, including regional and promotional details, sales data, and store attributes. By merging these tables, we aim to create a centralized resource that facilitates more comprehensive analyses, enables a holistic understanding of store performance, and enhances the accuracy of sales predictions through the integration of relevant factors.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 421570 entries, 0 to 421569
Data columns (total 16 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   Store         421570 non-null  object
 1   Dept          421570 non-null  object
 2   Date          421570 non-null  datetime64[ns]
 3   Weekly_Sales  421570 non-null  float64
 4   IsHoliday     421570 non-null  int64
 5   Type          421570 non-null  object
 6   Size          421570 non-null  int64
 7   Temperature   421570 non-null  float64
 8   Fuel_Price    421570 non-null  float64
 9   MarkDown1     421570 non-null  float64
 10  MarkDown2     421570 non-null  float64
 11  MarkDown3     421570 non-null  float64
 12  MarkDown4     421570 non-null  float64
 13  MarkDown5     421570 non-null  float64
 14  CPI           421570 non-null  float64
 15  Unemployment  421570 non-null  float64
dtypes: datetime64[ns](1), float64(10), int64(2), object(3)
memory usage: 51.5+ MB
```

2.3. Feature Extraction

Two valuable features can be derived from the Date column, namely the Year and Month. Extracting these features from the Date proves to be a viable approach, as they exhibit a strong correlation with the weekly sales patterns as shown in the figure below.



The inclusion of these Year and Month features in our model has yielded a noteworthy 3% improvement in performance as shown in the figure below. This outcome aligns logically with the understanding that sales exhibit discernible trends over time and are influenced by seasonal patterns. As a result, we have opted to retain these newly added features, Year and Month, within our model, recognizing their significant contribution to its predictive capacity.

```
R2 Score without Year,Month Features: 0.91
R2 Score with Year,Month Features: 0.94
```

2.4. Feature Selection

In our quest to identify the most effective feature selection approach, we employed two distinct models: mRMR (minimum Redundancy Maximum Relevance) and RFE (Recursive Feature Elimination). The goal is to assess how the number of selected features impacts the accuracy of a DecisionTreeRegressor model trained on Walmart sales data.

The first step in the code is to filter the dataset based on specific date ranges, effectively splitting the data into training and testing sets. The training set covers the period from February 2010 to March 2012, while the test set encompasses April 2012 to October 2012. Features and target values for both the training and testing sets are defined accordingly.

The code then creating an instance of the OrdinalEncoder class, which is commonly used for encoding categorical features into numerical values. It then fits this encoder to the feature matrix X, excluding the Date and Weekly_Sales columns, establishing a mapping of categories to numeric values. The original feature matrix X is subsequently transformed into new matrices, X_train_trans and X_test_trans, where categorical values are encoded as numeric values.

The code defines two functions, run_model_MRMR and run_model_greedy, both of which perform feature selection and model fitting:

1. **run_model_MRMR(k):** This function uses the mRMR method with a specified number of columns (k) to select the most relevant features. It employs SelectKBest to identify the top k features based on their relevance to the target variable Weekly_Sales using the f_regression scoring function. A DecisionTreeRegressor model is then trained on the selected features, and its R-squared score is computed. The results, including k and the R-squared score, are appended to the result array.
2. **run_model_greedy(k):** In this function, the RFE method is applied to select k features. Similar to the previous function, a DecisionTreeRegressor model is trained on the selected features, and the R-squared score is calculated.

The code then proceeds to execute both feature selection methods for various values of k, ranging from 2 and continues up to the total number of columns in the test dataset. The results are stored in a Pandas DataFrame, including the number of features (k), the R-squared score achieved using mRMR, and the R-squared score achieved using the RFE method.

Finally, the code generates a line plot that visualizes how the number of selected features affects the model accuracy for both mRMR and RFE. The x-axis represents the number of columns/features (k), and the y-axis shows the model accuracy. The plot helps in comparing the two feature selection methods' performance.

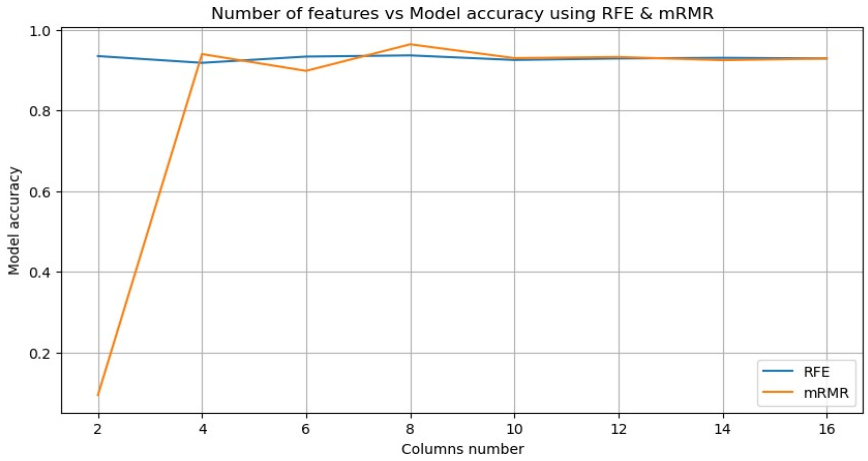
The Code:

```
# Filter the data for the training set (February 2010 to March 2012)
train_start_date = '2010-02-01'
```

```
train_end_date = '2012-03-31'
train_mask = (df_copy['Date'] >= train_start_date) & (df_copy['Date'] <=
train_end_date)
X_train = df_copy[train_mask].drop(columns=['Date', 'Weekly_Sales'])
y_train = df_copy[train_mask]['Weekly_Sales']

# Filter the data for the test set (April 2012 to October 2012)
test_start_date = '2012-04-01'
test_end_date = '2012-10-31'
test_mask = (df_copy['Date'] >= test_start_date) & (df_copy['Date'] <=
test_end_date)
X_test = df_copy[test_mask].drop(columns=['Date', 'Weekly_Sales'])
y_test = df_copy[test_mask]['Weekly_Sales']

# Create an instance of the OrdinalEncoder class, which is used for
encoding categorical features into numerical values.
model_incoder = OrdinalEncoder()

# Fit the OrdinalEncoder on the feature matrix X. This step determines the
mapping of categories to numeric values.
model_incoder.fit(df_copy.drop(columns=['Date', 'Weekly_Sales']))

# Transform (encode) the original feature matrix X into a new matrix
x_trans where categorical values are represented as numeric values.
X_train_trans = model_incoder.transform(X_train)
X_test_trans = model_incoder.transform(X_test)
def run_model_MRMR(k):
    '''
    Peform feature selection using the mRMR method, given a certain number
of columns.
    The function will append model accuracy with the result array
    '''
    global X_train_trans
    global X_test_trans
    slct_best = SelectKBest(k=k, score_func=f_regression)
    slct_best = slct_best.fit(X_train_trans, y_train)
    X_train_trans_ = slct_best.transform(X_train_trans)
    X_test_trans_ = slct_best.transform(X_test_trans)
    model = Model()
    model.fit(X_train_trans_, y_train)
    predictions = model.predict(X_test_trans_)
    r2 = r2_score(y_test, predictions)
    result.append({'k': k, 'MRMR R2': r2})

def run_model_greedy(k):
```

```
    '''
    Peform feature selection using the RFE method, given a certain number
of columns.
    The function will append model accuracy with the result array
    '''
    global X_train_trans
    global X_test_trans
    model = Model()
    rfe = RFE(model,n_features_to_select = k)
    rfe = rfe.fit(X_train_trans,y_train)
    X_train_trans_ = rfe.transform(X_train_trans)
    X_test_trans_ = rfe.transform(X_test_trans)
    model = Model()
    model.fit(X_train_trans_,y_train)
    predictions = model.predict(X_test_trans_)
    return r2_score(y_test,predictions)
Model = DecisionTreeRegressor
result = []
for k in range(2,len(X_test.columns)+1,2):
    run_model_MRMR(k)
result = pd.DataFrame(result)
result_greedy = []
for k in result['k']:
    result_greedy.append(run_model_greedy(k))
result['greedy'] = result_greedy
ax = plt.figure(figsize=(10,5))
plt.plot(result['k'],result['greedy'], label = 'RFE')
plt.plot(result['k'],result['MRMR R2'], label = 'mRMR')
plt.legend()
plt.title('Number of features vs Model accuracy using RFE & mRMR')
plt.xlabel('Columns number')
plt.ylabel('Model accuracy')
plt.grid()
plt.show()
```

The results unveiled an intriguing trend: as we increased the value of "k," the accuracy of the models consistently improved as illustrated in the figure below. This observation led us to a compelling conclusion - all the features within our dataset are inherently relevant to our sales prediction task. Therefore, we have made the decision to include all of these features when training our model.



Furthermore, we explored an alternative encoding technique known as ordinal encoder, a response to the complexity introduced by one-hot encoding, which generates a large number of sub-features or columns, making feature selection more challenging. The results were enlightening, revealing two key takeaways:

1. An increase in the number of selected columns directly corresponded to an improvement in model accuracy, reinforcing our decision to retain all features in our model.
2. Ordinal encoding emerged as a viable alternative to one-hot encoding, yielding a similar level of model accuracy while requiring significantly less computational resources. With ordinal encoding, the final number of features was reduced to 16, compared to the 153 generated by one-hot encoding.

Based on these findings, we have opted to leverage ordinal encoding in our model, as it strikes an optimal balance between accuracy and computational efficiency, ensuring a streamlined and effective feature selection process.

2.5. ML Model Selection and Training

The primary goal of this automation is to simplify the process of training and evaluating machine learning models for predicting weekly sales at Walmart stores and choose the most suitable model. To achieve this, we have developed two essential functions:

1. **train_model Function:** This function streamlines the entire modeling pipeline. It preprocesses the data by applying one-hot encoding to categorical columns and scaling numerical columns, combines the preprocessing transformers into a pipeline along with the specified model, fits the pipeline on the training data, measures the training time, and ultimately returns the trained machine learning model.
2. **evaluate_model Function:** The second function, uses the trained model to make predictions on the test features. It calculates key performance metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-squared (R^2), and finally returns a dictionary containing the computed evaluation metrics.

Several regression models are defined in a dictionary called models. These include Linear Regression, Ridge Regression, Lasso Regression, Decision Tree Regressor, and Random Forest Regressor.

A loop iterates through the defined regression models in the models dictionary. For each model, the train_model function is called to train the model using the training data. Subsequently, the evaluate_model function is used to evaluate the model's performance on the test data. Evaluation results, including MAE, RMSE, and R-squared, are stored in a dictionary called results. Finally, the code prints the evaluation results for each model, displaying metrics such as MAE, RMSE, and R-squared.

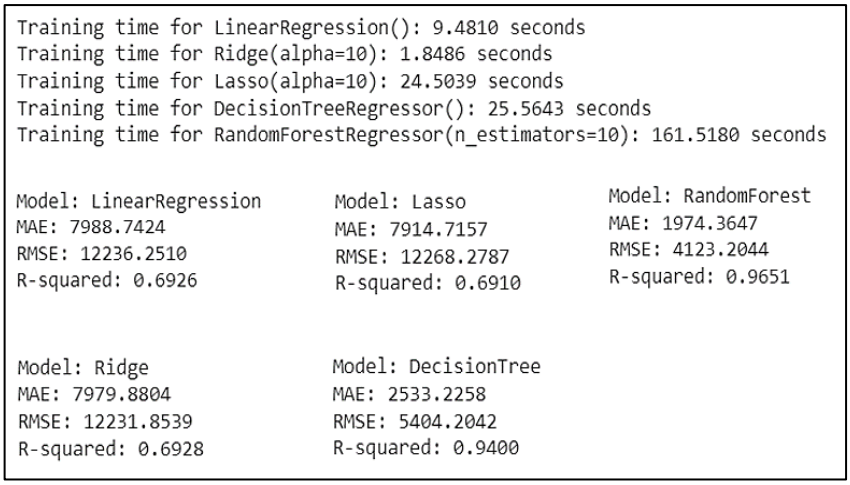
The Code:

```
def train_model(model, X_train, y_train):
    """
    Train a machine learning model with data preprocessing using a
pipeline.
    Parameters:
    - model: The machine learning model to be trained.
    - X_train: The training features.
    - y_train: The training target values.

    Returns:
    - trained_model: A trained model wrapped in a pipeline, including
preprocessing steps.
    This function takes a machine learning model, training features
(X_train), and training target values (y_train).
    It performs the following steps:
    1. Preprocesses the data by applying one-hot encoding to categorical
columns and scaling numerical columns.
    2. Combines the preprocessing transformers into a pipeline along with
the specified model.
    3. Fits the pipeline on the training data.
    4. Measures the training time and prints it.
    The trained model, including preprocessing steps, is returned for
further use.
    """
    # Extract and categorize the column data types into categorical and
numerical for further preprocessing.
    data_types = X_train.dtypes
    categorical_columns = data_types[data_types ==
'object'].index.tolist()
    numerical_columns = data_types[data_types != 'object'].index.tolist()
```

```
    # Create transformers for preprocessing categorical and numerical
columns
    categorical_transformer = Pipeline(steps=[('onehot',
OneHotEncoder(sparse_output=False))])
    numerical_transformer = Pipeline(steps=[('scaler', StandardScaler())])

    # Combine transformers for both types of columns
    preprocessor = ColumnTransformer(
        transformers=[
            ('cat', categorical_transformer, categorical_columns),
            ('num', numerical_transformer, numerical_columns)
        ], remainder='passthrough')


    #Create the final pipeline
    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', model)
    ])
    # Record the start time to measure the training duration of the
machine learning pipeline.
    start_time = time.time()

    # Fit the machine learning pipeline on the training data, measure the
training time, and print the duration.
    pipeline.fit(X_train, y_train)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Training time for {model}: {elapsed_time:.4f} seconds")
    return pipeline

# Function to evaluate a model
def evaluate_model(model, X_test, y_test):
    """
        Evaluate a machine learning model on a test dataset.
        Parameters:
        - model: The trained machine learning model to be evaluated.
        - X_test: The test features.
        - y_test: The true target values for the test dataset.
        Returns:
        - evaluation_results: A dictionary containing evaluation metrics,
including MAE, MSE, and R-squared.
        This function takes a trained machine learning model, test
features (X_test), and true target values (y_test).
        It performs the following steps:
```

```
        1. Uses the trained model to make predictions on the test
features.
        2. Calculates Mean Absolute Error (MAE), Mean Squared Error (MSE),
and R-squared (R2) between the predicted
           values and true target values.
        3. Returns a dictionary containing the computed evaluation
metrics.
         The evaluation results provide insights into the model's
performance on the test dataset.
    """
    # Use the trained model to make predictions on the test data.
    # Then calculate and return evaluation metrics including MAE, RMSE,
and R-squared.
    y_pred = model.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    rmse = mean_squared_error(y_test, y_pred,squared=False)
    r2 = r2_score(y_test, y_pred)
    return {"MAE": mae, "RMSE": rmse, "R-squared": r2}

# Define your models
models = {
    'LinearRegression': LinearRegression(),
    'Ridge': Ridge(alpha=10),
    'Lasso': Lasso(alpha=10),
    'DecisionTree': DecisionTreeRegressor(),
    'RandomForest':RandomForestRegressor(n_estimators=10)
}
# Train the models and store them in a dictionary
trained_models = {}
for model_name, model in models.items():
    trained_models[model_name] = train_model(model, X_train, y_train)

# Evaluate the models and store the results in a dictionary
results = {}
for model_name, model in trained_models.items():
    results[model_name] = evaluate_model(model,X_test, y_test)

# Print the results
for model_name, metrics in results.items():
    print(f"\nModel: {model_name}")
    for metric_name, value in metrics.items():
        print(f"{metric_name}: {value:.4f}")
    print('\n')
```

After we executed the code, the training time and evaluation metrics for each model appeared as shown in the figure.



Comparing the results, we decided to choose the decision tree model as the better performing model for forecasting Walmart's weekly sales based on several factors:

1. **Excellent Predictive Performance:** The Decision Tree model exhibits outstanding predictive performance, as indicated by its low MAE, RMSE, and high R^2 value. These metrics collectively suggest that the model captures and explains a significant portion of the variance in the data, resulting in accurate sales predictions.
2. **Faster Training Time:** The Decision Tree model boasts a relatively shorter training time compared to the Random Forest model. While it takes significantly less time to train, it still delivers competitive predictive performance, making it a practical choice for large datasets or scenarios where model training time is a concern.
3. **Low Complexity:** Decision Trees have a straightforward structure with a hierarchy of decisions based on features, making them less complex than ensemble methods like Random Forests. This simplicity can reduce the risk of overfitting.
4. **Applicability:** Decision Trees are versatile and can be easily adapted to handle various types of features and datasets. Their ability to handle both categorical and numerical data makes them suitable for a wide range of applications, including sales prediction for Walmart stores.

2.6. Hyperparameters Tuning

In this section, we detail the process of hyperparameter tuning for the Decision Tree Regressor model. The objective is to optimize the machine learning model's performance by exploring various hyperparameter combinations.

The code begins by defining a parameter grid to guide the hyperparameter tuning process. It specifies the following hyperparameters:

- **'criterion':** The criteria for splitting nodes, where we consider 'squared_error' and 'friedman_mse'.
- **'min_samples_split':** The minimum number of samples required to split an internal node in the tree, which includes 2, and 30.
- **'max_depth':** It specifies the maximum depth of the decision tree, with multiple depth values to explore, which include None, and 105.

Next, the OrdinalEncoder is initialized and will be used to encode categorical features into numeric values. The DecisionTreeRegressor model is then initialized, and this is the model that will be tuned using hyperparameter optimization.

The encoder is fitted on the training data, excluding non-numeric columns ('Date' and 'Weekly_Sales'). This step prepares the encoder to transform categorical features into numeric values. Then the X_train features are transformed using the fitted encoder (encoder). This step converts categorical features into numeric representations so that they can be used with the machine learning model.

Hyperparameter tuning is performed using GridSearchCV with cross-validation (5 folds), meaning the dataset will be divided into five parts, and the model's performance will be evaluated five times, each time with a different part used for validation. The grid of hyperparameters defined earlier (param_grid) is explored, and identify the optimal combination of hyperparameters by minimizing the negative mean squared error.

The grid_search.fit() method is called to execute the hyperparameter search. It performs a systematic search over the hyperparameter grid, training and evaluating the DecisionTreeRegressor with different combinations of hyperparameters.

The Code:

```
# Define a parameter grid for hyperparameter tuning
param_grid = {
    'criterion': ['squared_error','friedman_mse'],# Valid criteria for
regression
    'max_depth': [None, 105], # Maximum depth of the tree
    'min_samples_split': [2, 30] # Minimum samples required to split an
internal node
}
# Initialize an OrdinalEncoder for categorical feature encoding.
encoder = OrdinalEncoder()
# Initialize a DecisionTreeRegressor model.
model = DecisionTreeRegressor()
# Fit the OrdinalEncoder on the training data after excluding non-numeric
columns.
encoder.fit(df_copy.drop(columns=['Date', 'Weekly_Sales']))
# Transform the training features using the fitted encoder.
```

```
X_trans = encoder.transform(X_train)
# Hyperparameter tuning using GridSearchCV.
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
scoring='neg_mean_squared_error', cv=5)
# Execute the hyperparameter search.
grid_search.fit(X_trans, y_train)
```

Once the model is trained and tuned, it is serialized and saved to a file using the Pickle library. This serialized model file can be used for deployment in real-world applications, such as web deployment.

```
# Serialize and save the trained model (grid_search) using Pickle to a
file named 'model.pkl'.
import pickle
pickle.dump(grid_search, open('model.pkl', 'wb'))
```

2.7. ML Model Evaluation

This section focuses on model evaluation and visualization to assess the performance of a machine learning model that has undergone hyperparameter tuning. The code begins by retrieving the best estimator from the earlier performed GridSearchCV (grid_search.best_estimator_). Additionally, grid_search.best_params_ returns a dictionary containing the hyperparameters that resulted in the best performance during hyperparameter tuning, which are 'criterion': 'squared_error', 'max_depth': None, and 'min_samples_split': 30.

The variable grid_search.best_score_ represents the negative mean squared error achieved by the model with the optimal hyperparameters during the hyperparameter tuning process. Specifically, it has a value of approximately -197,807,670.28.

X_test_trans is created by transforming the test features (X_test) using the same encoder (encoder) that was used to preprocess the training data. y_pred contains the predictions made by the best estimator (model with optimal hyperparameters) on the test data.

r2_score(y_test, y_pred) calculates the R-squared score on the test dataset to assess how well the model's predictions (y_pred) align with the true target values (y_test). The R-squared score, approximately 0.9415, indicates a strong correlation between the model's predictions and the actual target values on the test data, suggesting that the model performs well on unseen data.

The Code:

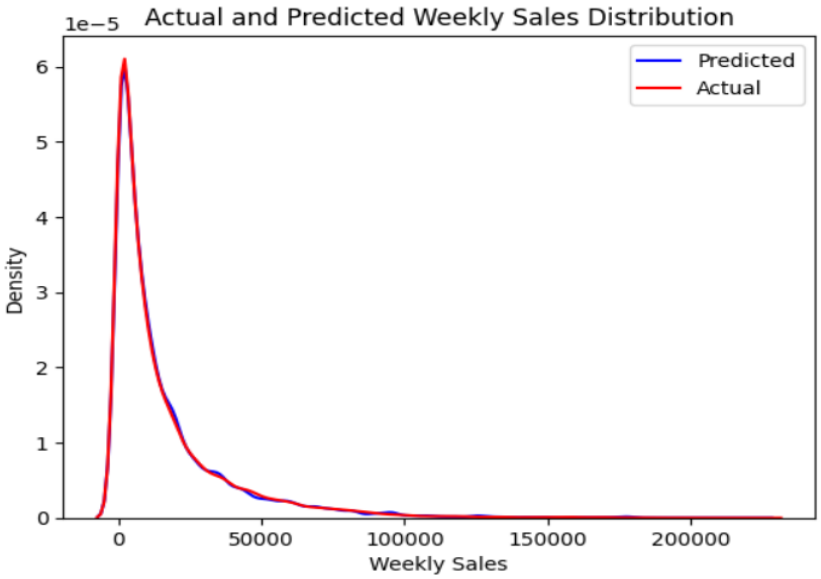
```
# Retrieve the best estimator (model with optimal hyperparameters) from
the GridSearchCV.
grid_search.best_estimator_
# Get the best hyperparameters selected by GridSearchCV.
```

```
grid_search.best_params_
# Retrieve the negative mean squared error achieved during hyperparameter
tuning.
grid_search.best_score_
# Make predictions using the best estimator from GridSearchCV and
calculate the R-squared score on the test data.
X_test_trans = encoder.transform(X_test)
y_pred = grid_search.predict(X_test_trans)
r2_score(y_test,y_pred)
```

The code uses the Seaborn library to create a Kernel Density Estimation (KDE) plot for model predictions (y_pred) and actual target values (y_test) as shown in the figure below. KDE plots are valuable for visually assessing the alignment and accuracy of the model's predictions compared to the actual data. Predicted and actual distributions are depicted using different colors (blue for predicted and red for actual), facilitating a quick evaluation of how well the model captures underlying patterns.

The Code:

```
# Create Kernel Density Estimation (KDE) plot for model predictions
(y_pred).
sns.kdeplot(y_pred,color='b',label='Predicted')
# Create a separate KDE plot for actual target values (y_test).
sns.kdeplot(y_test,color='r',label='Actual')
plt.legend()
plt.xlabel('Weekly Sales')
plt.title('Actual and Predicted Weekly Sales Distribution')
plt.show()
```



2.8. Feature Importance

We used the Yellowbrick library to create a "FeatureImportances" visualization. This visualization is instrumental in evaluating the importance of individual features within the model for predicting weekly sales. Understanding feature importance helps identify which variables have the most significant impact on the model's predictions.

The visualization is configured with several key parameters:

- grid_search.estimator: Specifies the machine learning model under evaluation, which is part of the previously trained and tuned pipeline.
- labels=X_train.columns.values: Sets the labels for features, extracted from the training data (X_train), to be displayed in the visualization.
- topn=5: Limits the visualization to show the top 5 most important features, making it easier to focus on the most impactful variables.
- colors='b': Determines the color of the feature importance bars, with blue used in this instance.

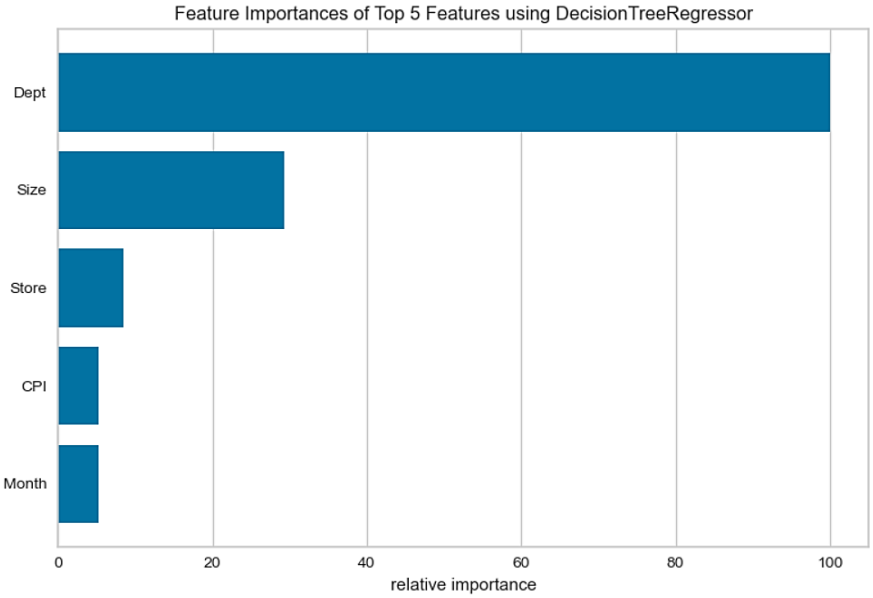
The visualization is fitted to the training data (X_trans) and the corresponding target variable (y_train). During this step, the model assesses the importance of each feature based on the training data.

Finally, the visualization is displayed using viz.show(). This visual representation provides insights into which features contribute the most to the model's predictions, aiding in feature selection, interpretation, and model refinement.

The Code:

```
from yellowbrick.model_selection import FeatureImportances
# Create a FeatureImportances visualization using the Yellowbrick library
to assess feature importance.
viz = FeatureImportances(grid_search.estimator,
labels=X_train.columns.values, topn=5, colors='b')
# Fit the visualization to the training data and the target variable.
viz.fit(X_trans, y_train)
# Show the visualization
viz.show()
plt.show()
```

Notably, our visualization pinpointed that the Dept and Size features wield significant importance in our predictive model, as exemplified in the accompanying figure.



2.9. SARIMA Model

Introduction to Time Series Forecasting

Time series forecasting is a critical area of data analysis that deals with making predictions about future values based on historical time-ordered data points. Time series data consists of observations recorded at successive, evenly spaced time intervals. This data can be found in various domains, such as finance, economics, weather, sales, and more, making time series forecasting an essential tool for decision-making and planning in many fields.

Key Characteristics of Time Series Data:

1- **Time Dependency**: Time series data exhibits a clear chronological order, where each data point is associated with a specific time or date. The temporal ordering is a fundamental characteristic that distinguishes time series data from other types of data.
2- **Temporal Patterns**: Time series data often contains patterns, trends, and seasonality. Patterns represent recurring behaviors or trends in the data, while seasonality reflects regular, repeating fluctuations influenced by calendar or seasonal factors.
3- **Noise**: Time series data can also contain random fluctuations or noise, which can make forecasting challenging. Distinguishing between genuine patterns and noise is a key task in time series analysis.

Objectives of Time Series Forecasting:

The primary goal of time series forecasting is to predict future values or trends in a time series dataset. This predictive analysis serves various purposes, including:

1- **Decision-Making**: Forecasting provides valuable insights for making informed decisions in various domains. For example, businesses use sales forecasts to optimize inventory management and production planning.

2- **Resource Allocation**: Accurate forecasts help allocate resources efficiently. For instance, energy companies use forecasts to plan electricity generation and distribution.

3- **Risk Management**: Forecasting assists in assessing and mitigating risks. Financial institutions use market forecasts to manage investments and reduce exposure to market volatility.

Comparison between ARIMA and SARIMA Models:

ARIMA (Autoregressive Integrated Moving Average):

Definition: ARIMA is a statistical method used for time series forecasting. It stands for Autoregressive Integrated Moving Average.

Purpose: ARIMA is designed to capture different aspects of a time series data: autocorrelation (AR), differencing to achieve stationarity (I), and moving average (MA) components. It is used for forecasting univariate time series data, where past observations are used to predict future values. The ARIMA model can be represented as ARIMA(p, d, q), where 'p' is the order of the autoregressive component, 'd' is the degree of differencing required to make the time series stationary, and 'q' is the order of the moving average component.

Differences from SARIMA: ARIMA models do not consider seasonality explicitly. They are suitable for non-seasonal time series data, and if there is a strong seasonal component, ARIMA may not perform well. SARIMA extends ARIMA by adding seasonal components, making it better suited for time series data with seasonality.

SARIMA (Seasonal Autoregressive Integrated Moving Average):

Definition: SARIMA is an extension of ARIMA that includes seasonal components. It stands for Seasonal Autoregressive Integrated Moving Average.

Purpose: SARIMA is used when dealing with time series data that exhibit seasonal patterns, which are recurring patterns that follow a fixed time interval (e.g., monthly, quarterly, yearly). SARIMA models capture both non-seasonal and seasonal components in the data. The SARIMA model can be represented as SARIMA(p, d, q)(P, D, Q, S), where 'p', 'd', and 'q' are the non-seasonal ARIMA components, and 'P', 'D', 'Q', and 'S' are the seasonal ARIMA components.

Differences from ARIMA: SARIMA explicitly accounts for seasonal patterns in addition to the non-seasonal patterns considered by ARIMA. It is more suitable for time series data with clear seasonal variations.

Code Review and Analysis: SARIMA Time Series Forecasting:

Below is a detailed report analyzing the provided code for SARIMA time series forecasting of weekly sales data for a specific store. The report covers various aspects of the code, including its structure, and functionality.

Code Structure and Organization:

The code is structured as a Python function named forecast, which takes several parameters to perform SARIMA forecasting.

It follows a clear structure with comments explaining each section of the code.

Function Signature and Parameters:

The function uses a keyword-only argument syntax, which is a good practice for ensuring clear and error-resistant function calls.

Parameters include DataFrame, Store, Month, and Year, which are essential for specifying the store and time period for forecasting.

Data Preprocessing:

The code starts by filtering the input DataFrame to select data for the specified store, which is a necessary step in time series analysis.

Training and Test Data Splitting:

The code splits the time series data into training and test sets. However, the splitting is currently based on fixed indices (120 and 23). Making these values configurable or data-driven would be more flexible.

SARIMA Model Implementation:

It defines SARIMA model parameters (order and seasonal_order) for model configuration.

The SARIMA model is created and fitted to the training data using the SARIMAX class from the statsmodels library.

Forecasting:

The code generates forecasts for a specified period using the trained SARIMA model.

It filters the forecasted values for the specified Month and Year, allowing for customized predictions.

Bias Calculation and Testing:

Bias in the forecast is calculated by comparing the predicted values to the observed values (forecast errors).

A t-test is performed to determine if the bias is statistically significant, providing insights into the accuracy of the model's forecasts.

Visualization:

The code produces a graphical visualization of observed and forecasted sales, making it easier to assess model performance visually.

Forecast Bias:

Forecast bias refers to a consistent tendency in a forecasting model to either overestimate (positive bias) or underestimate (negative bias) actual values. In simpler terms, it means the model consistently gets predictions either too high or too low.

Understanding forecast bias is essential because it affects decision-making and resource allocation. Overcoming bias involves refining the model, adjusting parameters, or improving data quality to make forecasts more accurate. Monitoring bias helps maintain forecast reliability.

- Forecast errors are calculated as the difference between the observed sales and predicted sales.
- The code calculates the mean of these forecast errors, which represents the bias in the forecasts.
- A t-test is performed to check if the bias is statistically significant. If the p-value from the test is less than 0.05, it indicates a significant bias.

The code:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
df_copy.Date= pd.to_datetime(df_copy.Date)
df_time_series= df_copy[['Date','Store','Weekly_Sales']]
df_time_series.to_csv('time_series.csv',index=False)
```

```
def forecast (*,DataFrame,Store,Month,Year):
    """
    Perform SARIMA time series forecasting for a specific store's weekly
sales.
    The model will forecast results between 2012-05-18 and 2014-01-03. If
an inserted date was before 2012-05-18,
    the function's average weekly sales result will be historical and not
forecasted.

    Parameters:
    - DataFrame (pd.DataFrame): The input DataFrame containing store data.
    - Store (int): The store identifier for which forecasting is
performed.
    - Month (int): The target month for which sales forecasting is
desired.
    - Year (int): The target year for which sales forecasting is desired.

    Returns:
    float: The mean of weekly sales for the specified Month and Year.

    This function filters the input DataFrame to select data for the
specified store,
    calculates the mean weekly sales, and then uses a SARIMA model to
forecast sales for
    a specified period into the future. It prints the mean weekly sales
for the specified
    month and year, the R-squared score, and visualizes the results. If no
forecasted
    values are available for the specified month and year, it falls back
to using the
    training data and prints a message.
    """
     # Filter data for the specified store
    df_forecast= DataFrame[(DataFrame.Store==Store) ]

    # Calculate the mean weekly sales by grouping data by date
    ts=df_forecast.groupby('Date')['Weekly_Sales'].mean()
```

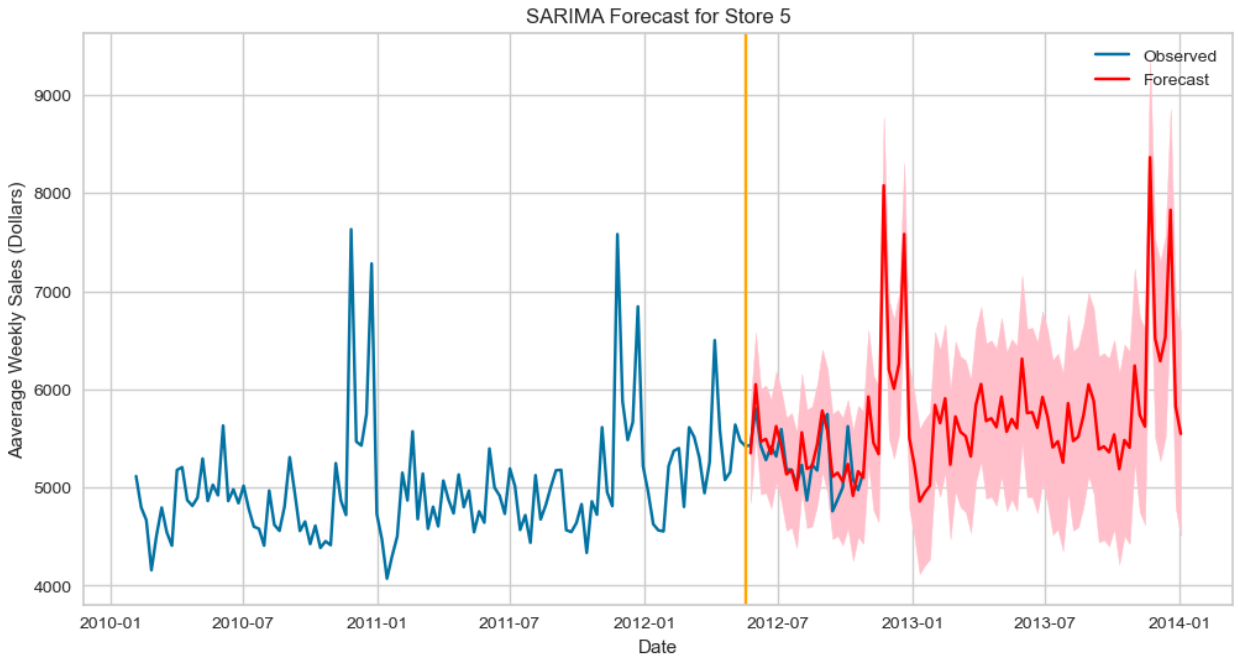
```
    # Split data into training and test sets
    ts_train= ts.head(120)
    ts_test= ts.tail(23)

    # Define SARIMA model parameters
    order = (1, 1, 1)
    seasonal_order = (1, 1, 1, 52)

    # Create and fit the SARIMA model to training data
    sarima_model = SARIMAX(ts_train, order=order,
seasonal_order=seasonal_order)
    sarima_results = sarima_model.fit()

    # Specify the forecast period
    forecast_period = 85

    # Generate forecasts for the specified period
    forecast = sarima_results.get_forecast(steps=forecast_period)
    forecasted_values=forecast.predicted_mean

    # Filter forecasted values for the specified Month and Year
    filtered=forecasted_values[(forecasted_values.index.year==Year) &
(forecasted_values.index.month==Month)]

    # If no forecasted values are available, use training data and print a
message
    if filtered.empty:
        filtered = ts_train[(ts_train.index.year==Year) &
(ts_train.index.month==Month)]
        print('Obsereved Values Not Forecasted')

    # Calculate R-squared score between observed and predicted sales
    forecast_conf_int = forecast.conf_int()
    ts_pred=[]
    for i in ts_test.index:
        ts_pred.append(sarima_results.predict(pd.Timestamp(i)))
    ts_pred=np.array(ts_pred)

    # Checking Forecast Bias
    # Check if the bias is significantly different from zero using a t-
test
    # You can perform a hypothesis test to determine statistical
significance.
    # A bias close to zero indicates that the model's forecasts are, on
average, accurate.
```

```
    forecast_errors = ts_test.to_numpy().reshape(ts_test.shape[0],1) -
ts_pred
    bias= np.mean(forecast_errors)
    t_stat, p_value = stats.ttest_1samp(forecast_errors, 0)

    if p_value < 0.05:  # Choose an appropriate significance level (e.g.,
0.05)
        print("The forecast has a significant bias.")
    else:
        print("The forecast does not have a significant bias.")

    # Print the mean Weekly sales for the specified Month and Year
    print('\n')
    print(f'The mean Weekly sales for month {Month} and year {Year} is
{filtered.mean():.2f}$')
    print(f'The R-Squared for Test set is equal to
{r2_score(ts_test,ts_pred):.3f}')


    # Visualize the results using Matplotlib
    plt.figure(figsize=(12, 6))
    plt.plot(ts, label='Observed')
    plt.plot(forecast.predicted_mean, label='Forecast', color='red')
    plt.fill_between(forecast_conf_int.index, forecast_conf_int.iloc[:,
0], forecast_conf_int.iloc[:, 1], color='pink')
    plt.axvline(ts_train.tail(1).index,color='orange')
    plt.title('SARIMA Forecast for Store {}'.format(Store))
    plt.ylabel('Aaverage Weekly Sales (Dollars)')
    plt.xlabel('Date')
    plt.legend()

    plt.show()




    # Return the mean of the filtered forecasted values
    return filtered.mean()
```

Running an example:

```
forcasted=forecast(DataFrame=df_copy,Store=5,Month=1,Year=2013)
```

```
The forecast does not have a significant bias.


The mean Weekly sales for month 1 and year 2013 is 5011.95$
The R-Squared for Test set is equal to 0.456
```



Sales Forecast Summary:

- ARIMA and SARIMA models are both used for time series forecasting.
- ARIMA models capture non-seasonal patterns, while SARIMA models capture both non-seasonal and seasonal patterns.
- The `forecast` function is used to perform time series forecasting for a specific store's weekly sales data. It utilizes the SARIMA (Seasonal Autoregressive Integrated Moving Average) model to generate forecasts and provide insights into the sales data. Here's a high-level summary of the process:

1. Data is filtered to select information for the specified store.
2. Mean weekly sales are calculated based on date grouping.
3. The data is split into a training set and a test set.
4. SARIMA model parameters are defined and the model is fitted to the training data.
5. Forecasts are generated for a specified future period.

6. Forecasted values are filtered based on the desired month and year.

7. If no forecasted values are available, observed values from the training data are used.

8. The mean weekly sales for the specified month and year are calculated and printed.

9. The R-squared score between observed and predicted sales is displayed.

10. A visualization shows observed and forecasted sales data.

Additional Bias Checking (Model Evaluation):

- After forecasting, the function checks for forecast bias by calculating forecast errors and performing a hypothesis test to determine statistical significance.

- A bias close to zero indicates that the model's forecasts are, on average, accurate.

- If a significant bias is detected, it indicates that the model may consistently overestimate or underestimate sales values.

This process provides valuable insights into future sales trends for a specific store.

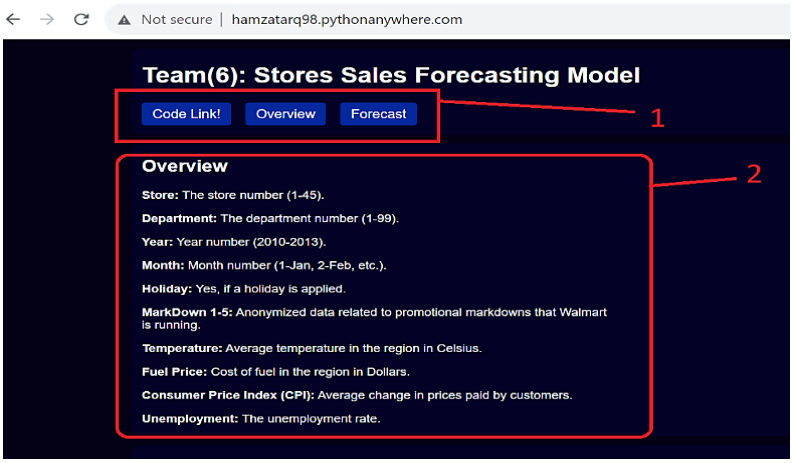
Chapter 3
Results & Conclusion

This chapter shows the method and technology we choose to present our work and how to implement the model by users, as well as brief summary for challenges we encountered during development phase, finally we will end chapter 3 with project conclusion and how our project is expected to deliver value for the end user.

3.1. ML Model Deployment

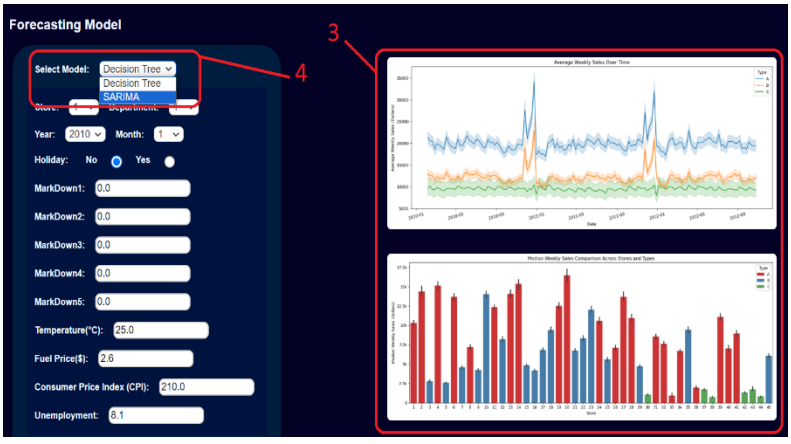
We decided to Implement our forecasting model into a web-site created via html and CSS codes and hosted by flask Framework and uploaded into the cloud so everyone can have access to it, follow the next link to get for the project web-site (link).

1) **Header section:** we can see three buttons, the first one will direct us for whole project code stored in a Git hub repository, and the last 2 buttons direct us for the web page sections (Overview and Forecast).



2) **Overview:** in the Overview section we can find detailed information about each most important features that the model is going to accept in order to produce prediction (Avg Weekly Sales).

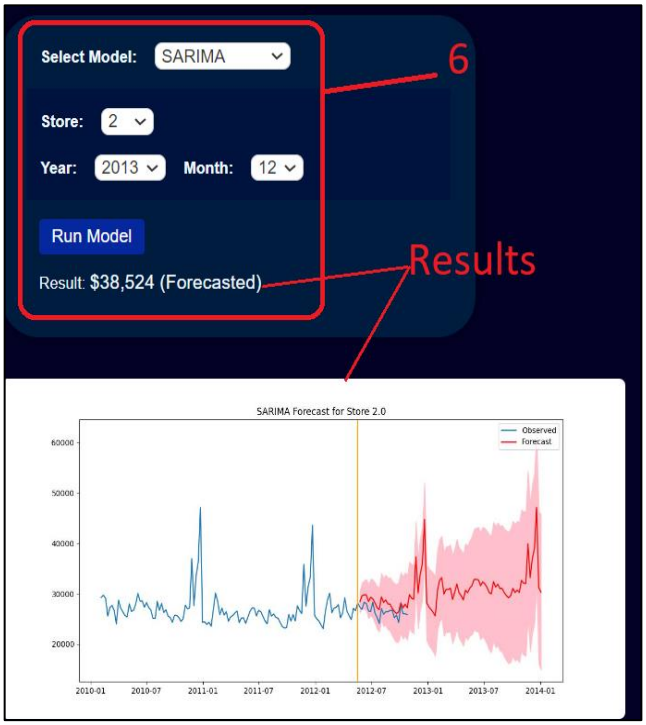
3) **Figures:** In the right 2 figures shows the most important insights in order to understand the actual data, first plot is a time series shows the change of avg weekly sales throughout the year for each store types (A, B and C), and second figure shows the Sales volumes for each Store Individually.



4) **Model Selection:** In this tab the user is allowed to choose one of two prediction models (Decision Tree, SARIMA) each model will require different input based on the needed data to run the model.

5) **Decision Tree Model:** This model will predict the weekly sales for each chosen individual Department in certain Store in particular conditions (Temperature, Holiday, Markdowns, …etc), also the avg actual numbers will be also displayed so we can compare between the actual and forecasted values

6) **SARIMA Model:** We built the SARIMA model to predict the avg sales on the Stores level based on time series data only, after running the model a time series plot will be displayed below the form showing the actual sales for particular Store in blue, and the forecasted data calculated by the SARIMA model in red,



3.2. Challenges

We ran into many challenges during the implementation phase below are most important ones:

- **Web development Knowledge**: It was our first attempts to develop a web-site or writing html, CSS to build a web page, with few searching and studying we managed to come up with the above web-site shown in the ML Model Deployment section.
- **Web Server not working on the cloud:** Many times we encountered the issue of the web server is working correctly in the local host but when pull it into the cloud it stopped working or start to get errors, after troubleshooting there are two main reasons for the web server not to work in the could while it function in the local PC:
    - First reason is due to the cloud reserved **disk space** we resolved this issue by simply increase the space allocated for our web-app in the cloud.
    - Framework and library **versions mismatch**, is the second reason for the web-server to malfunction on the cloud, the issue was solved by ensuring all of the tools' version used when developing the site locally is the same as the tools in the cloud (Libraries and Frameworks).

3.3. Conclusion

We believe our Model can benefit the user by the following:

- Study the chance of opening branches or departments based on upcoming forecasted sales.
- Make use of Holidays to boost Sales by applying special events and promotions and the sales are increased slightly in Holidays weeks.
- Achieve better control over Stores resources by considering Economic Indicators, Climate Change, Seasonality and Store characteristics while predicting Weekly Sales.