

2024 Monthly Retail/Food Sales Prediction and Forecasting

By Zohd Khan

In this personal project, I develop and employ a **Ridge Regression** algorithm to predict the **average monthly sales value (in millions)** across various types of businesses in America from January to May of 2024.

I then use the same algorithm to perform a **forecasting** of average monthly sales from June 2024 to June 2025 (with only data from January 2014 to May 2024).

I acquired the relevant data from the [US Census Bureau](#), with the source data file being titled "Retail and Food Services Sales (1992-Present)".

I also applied various Data Science techniques, such as **seasonal decomposition, cross validation, and algorithmic comparison**, to aid in the development and testing of my model.

Throughout the entire coding process, which is provided below, I **documented** the most significant decisions and analyses that I made while in pursuit of a well-performing model.

By the end of the process, I finalized an **exceptionally accurate** model that, with a RMSE score of 25.16 on my test data, was able to make predictions that **deviated only .018% from the true mean sales values, on average**. The average RMSE score across 5 folds in Cross-validation was also reasonably low (588.38), indicating the **absence of significant overfitting**.

Data Loading, Conversion, Storage and Processing

```
In [11]: import pandas as pd #Importing "pandas" to perform complex data handling operations
import numpy as np #Importing "numpy" to perform any essential mathematical operations

#Initializing file path. I specifically intend to use spreadsheets containing monthly sales data
file_path = "C:/Users/Zohd/OneDrive/Desktop/Personal Project 2/mrtssales92-present.xlsx"

#Initializing function which Loads each of the 10 spreadsheets, excludes unnecessary rows, and concatenates them into a single DataFrame
def sheets_to_csv(sheet_name):
    df = pd.read_excel(
        file_path,
        sheet_name = sheet_name,
        skiprows = [0,1,2] + list(range(5,72)) + list(range(111,123)), #Skipping irrelevant header rows
        header = None #Setting "header" argument to None since I am about to manually set headers
    )

    # Manually setting headers by taking indexing first two rows, altering NaN values, joining the 2 rows, stripping whitespaces, and converting everything to a list
    headers = df.iloc[0:2].fillna('').astype(str).apply(' '.join).str.strip().tolist()
    df = df[2:]
    df.columns = headers
    return df
```

```

#Setting the names of the columns of the dataframes as the headers manually set
df.columns = headers

#Dropping the old first 2 rows as they are no longer needed
df = df.drop([0,1])

#Setting up the csv file path for every single sheet that is converted
csv_file_path = f"C:/Users/Zohd/OneDrive/Desktop/Personal Project 2/{sheet_name}

#Using "to_csv" function to convert each sheet to csv file
df.to_csv(csv_file_path, index = False)
#print(f"Sheet {sheet_name} successfully converted to CSV.")

# Creating the list of sheet names (matching actual names in xlsx file) that will be used
sheet_names = ['2014', '2015', '2016', '2017', '2018', '2019', '2020', '2021', '2022']

#Looping through previous list and applying "sheets_to_csv" function create the new dataframes
for sheet_name in sheet_names:
    sheets_to_csv(sheet_name)

#print("Conversion from Excel sheets to CSV files completed.")

#Now that csv files are created, writing a separate function to load them
def load_csv(file_name):
    df = pd.read_csv(file_name)

    return df

#Assigning each of the new csv files an associated csv file path in order to load them
csv_file_paths = [f"C:/Users/Zohd/OneDrive/Desktop/Personal Project 2/{sheet_name}_retail.csv"]

#Applying load_csv function to each csv_file_path and storing the result in retail_data_csv
retail_data_csv = {sheet_name: load_csv(csv_file_path) for sheet_name, csv_file_path in zip(sheet_names, csv_file_paths)}

#Creating function to transpose each individual DataFrame, add a 'Year' column to keep track of the year
def reshape_df(df, year):
    df = df.set_index('Kind of Business').T # Transposing the DataFrame
    df['Year'] = year # Adding a Year column
    df = df.reset_index().rename(columns={'index': 'Month'}) # Resetting index and renaming columns
    df = pd.melt(df, id_vars=['Year', 'Month'], var_name='Kind of Business', value_name='Sales')
    return df

#Creating retail data list
retail_data_list = []

#Inspecting the first few rows of each dataframe by looping through "retail_data_csv"
for sheet_name, df in retail_data_csv.items():
    #Dropping the "NAICS Code" column as it is useless
    df = df.drop(columns = ["NAICS Code"])
    #Dropping all rows that are completely filled with NA values
    df = df.dropna(axis = 0, how = 'all')
    #Dropping all columns that are completely filled with NA values
    df = df.dropna(axis = 1, how = 'all')
    #Creating BOOLEAN MASK to check for presence of suppressed values "(S)" in each row
    suppressed_values = df.isin(['(S)').any(axis = 1)

```

```

#Subsetting each dataframe to select only values that don't contain (S) using "
df = df[~suppressed_values]
#Applying the reshape function from earlier to
reshaped_df = reshape_df(df, sheet_name)
#
retail_data_list.append(reshaped_df)
#Assigning all sheets with the changes I've made through the "df" variable
retail_data_csv[sheet_name] = df

#Concatenating the various dataframes within the "retail_data_csv" dictionary into
retail_data = pd.concat(retail_data_list, ignore_index=True)

#Resetting index to ensure Python does not get confused with old one
retail_data.reset_index(drop = True, inplace = True)

# Ensuring correct data types
retail_data['Year'] = retail_data['Year'].astype(str)
retail_data['Month'] = retail_data['Month'].str[:3]
month_mapping = {'Jan': '01', 'Feb': '02', 'Mar': '03', 'Apr': '04', 'May': '05', 'Jun': '06',
retail_data['Month'] = retail_data['Month'].map(month_mapping)
retail_data['Sales'] = pd.to_numeric(retail_data['Sales'], errors='coerce')

# Displaying the concatenated DataFrame shape to ensure all processing was successful
print("Shape of the concatenated DataFrame:", retail_data.shape)

#Printing the retail dataframe, and verifying that all NaN values are gone
print(retail_data.head(6))
print(retail_data.isnull().any())

#Getting the average sales value for each month of each year, by applying "groupby"
# businesses for each month
sales_data_aggregated = retail_data.groupby(['Year', 'Month']).agg({'Sales': 'mean'})

```

Shape of the concatenated DataFrame: (4656, 4)

| | Year | Month | Kind of Business | Sales |
|---|------------------|-------|---------------------------------------|----------|
| 0 | 2014 | 01 | Retail and food services sales, total | 419015.0 |
| 1 | 2014 | 02 | Retail and food services sales, total | 424793.0 |
| 2 | 2014 | 03 | Retail and food services sales, total | 429132.0 |
| 3 | 2014 | 04 | Retail and food services sales, total | 433530.0 |
| 4 | 2014 | 05 | Retail and food services sales, total | 434248.0 |
| 5 | 2014 | 06 | Retail and food services sales, total | 435438.0 |
| | Year | | False | |
| | Month | | False | |
| | Kind of Business | | False | |
| | Sales | | False | |
| | dtype: | bool | | |

This initial section of my code performs the loading, conversion, and storage of the 11 excel spreadsheets to respective CSV files, with each CSV file being associated with a year from **2014 to 2024**. The code then performs a significant amount of data processing and cleaning in order to prepare the data for analysis.

To complete this section, I first focused on removing any excel rows that contained unnecessary information, such as certain headers or comments, from each of the 11

spreadsheets. I decided to only keep the rows containing **adjusted** sales values since those values provide the most insight. I also fixed the formatting of any desired headers, and implemented the logic to assign a csv file to each spreadsheet. I loaded each csv file to a dataframe, and then placed all the dataframes into a **dictionary** object, with the keys and values being the years in the "sheet_name" list, and the corresponding dataframes, respectively.

I next reshaped each of the dataframes, specifically by transposing them (switching rows and columns), adjusting their columns, and changing their format from wide to long, all of which helped to maintain a consistent structure and ease future indexing.

After removing NA and suppressed values, I concatenated (linked) all the cleaned dataframes together to create a new, singular dataframe. I then applied various techniques to my time-related columns, all of which helped organize my data properly.

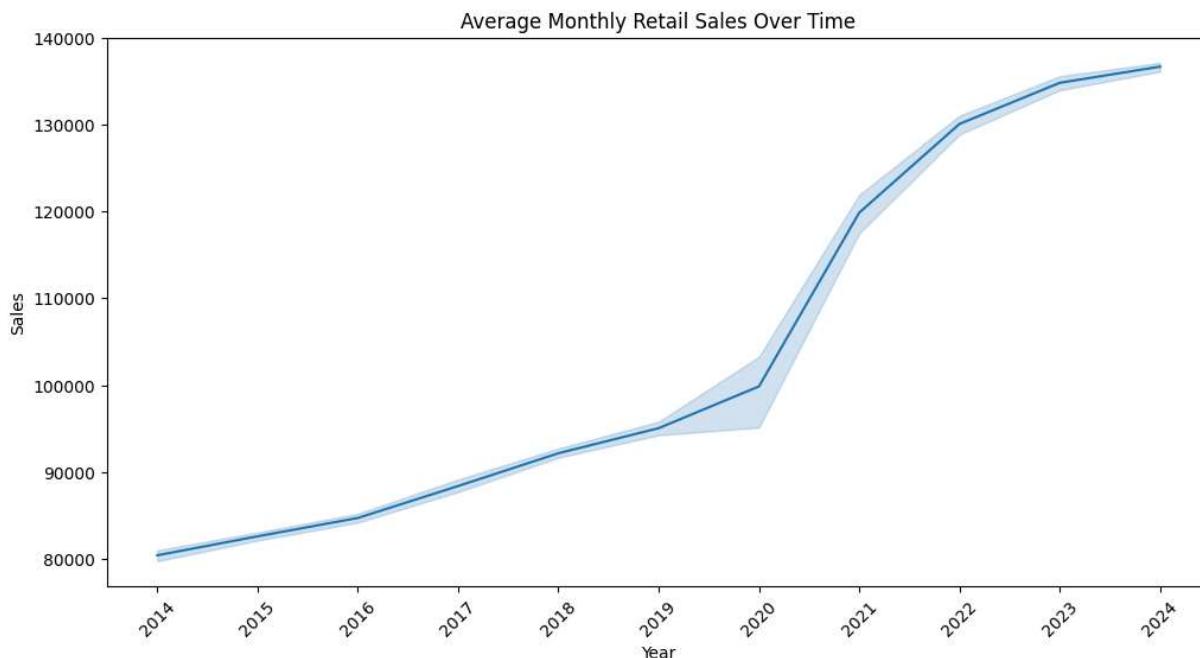
I finished this section by calculating the **average sales value for each month of each year**. To achieve this, I grouped the data by the 'Year' and 'Month' columns, and then calculated the mean sales value for each group, which provided a clear view of the average monthly sales trends across all businesses. The result was stored in a new dataframe, 'sales_data_aggregated', which I would work with for the remainder of the procedure, as this granularity of the data would be most convenient for prediction.

Data Visualization

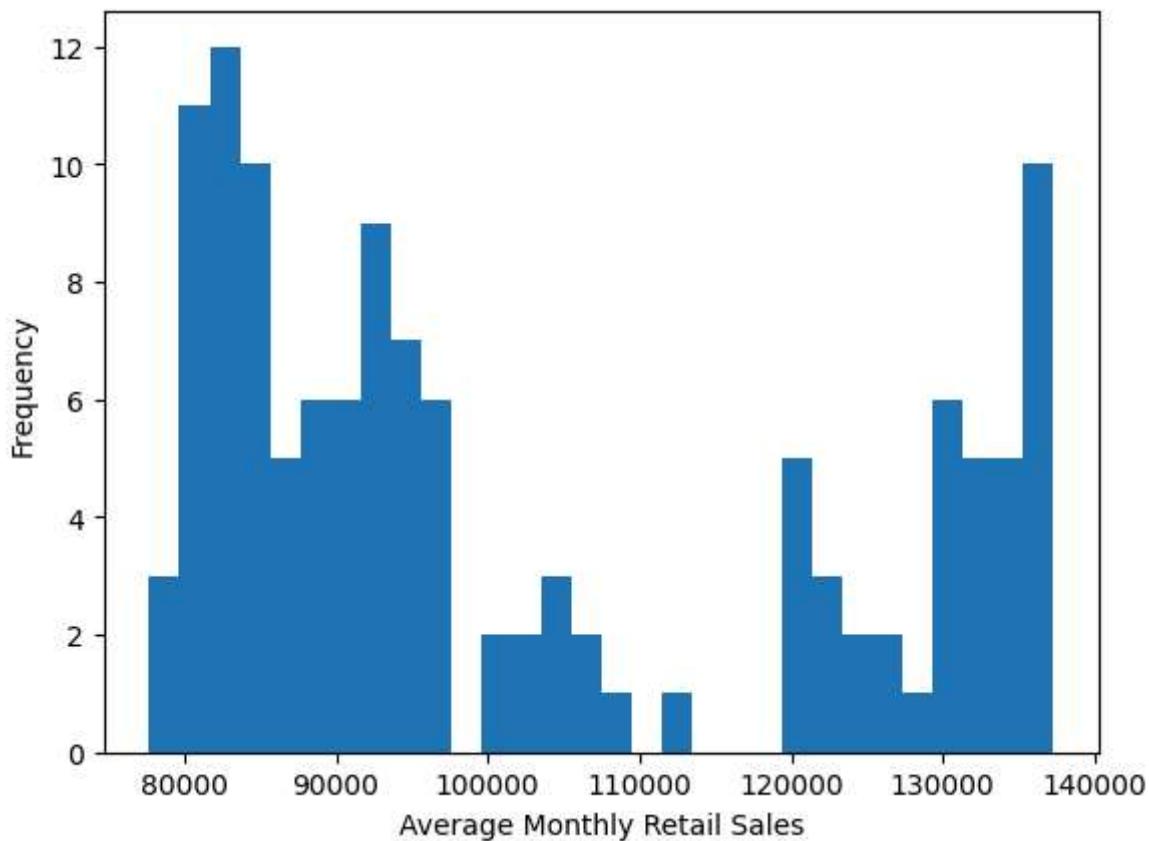
```
In [12]: import seaborn as sns #importing seaborn to use "Lineplot" function to plot Line de
import matplotlib.pyplot as plt #Using matplotlib for the figure details

#Setting up figure size with width of 12 and Length of 6
plt.figure(figsize = (12,6))
#Creating Line graph based on aggregated data with years on the x axis and sales on
sns.lineplot(data = sales_data_aggregated, x = 'Year', y = 'Sales')
plt.title('Average Monthly Retail Sales Over Time')
plt.xticks(rotation = 45)
plt.show()

sales_data_aggregated['Sales'].plot(kind = 'hist', bins = 30, xlabel = 'Average Mon
#print(sales_data_aggregated)
```



```
Out[12]: <Axes: xlabel='Average Monthly Retail Sales', ylabel='Frequency'>
```



This next portion of my code, despite being quite short, still provided some insight into the nature of the processed sales data. Specifically, the line plot shows that the sales across all businesses **generally increased** over the course of years from 2014 to 2024.

Furthermore, the histogram reveals a **multimodal distribution** to the data, indicating that the data may be clustered at relatively tight ranges.

When I first noticed this multimodal distribution, I speculated that it may be due to **seasonal effects**, which could cause similar sales values to appear during the same months across each year.

In order ascertain whether or not my suspicion was true, I decided to conduct a seasonal decomposition in the next section of the code.

Seasonal Decomposition

```
In [13]: #Importing module that enables execution of seasonal decomposition,which provides i
from statsmodels.tsa.seasonal import seasonal_decompose

#Creating new 'date' column that includes concatenation of year and month with hyph
#expects this. Also, the reason I am creating this column is to provide a consisten
sales_data_aggregated['Date'] = pd.to_datetime(sales_data_aggregated['Year'] + "-"

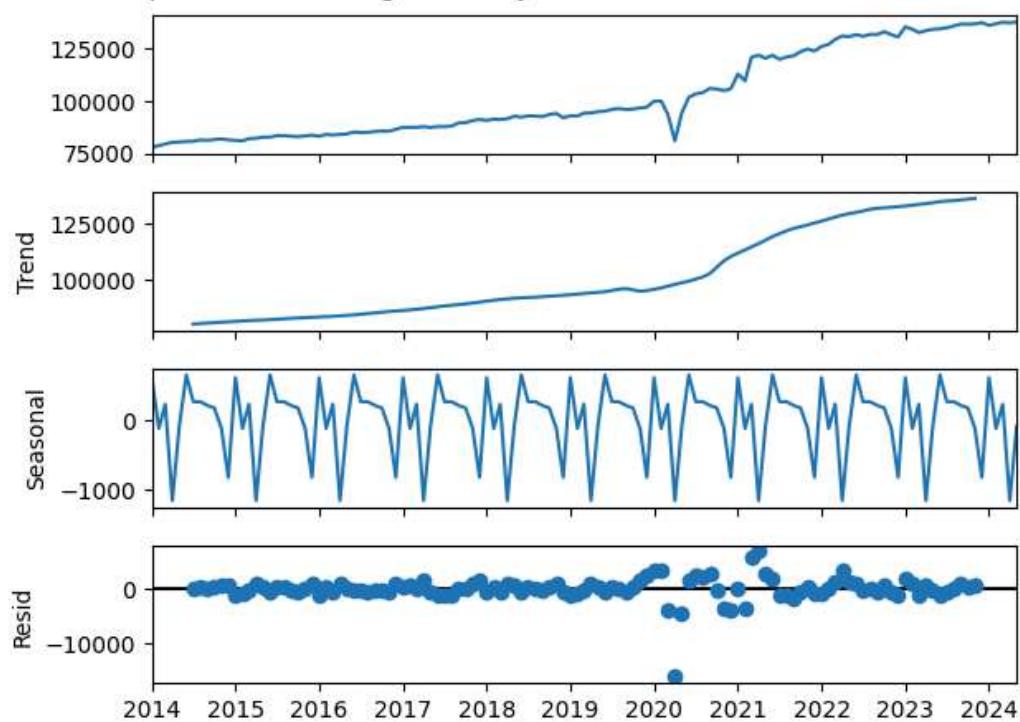
#Setting the index of the entire "sales_data_aggregated" dataframe to be based on t
#to be indexed by a certain time interval, with date being appropriate. Also, 'Date
#that is useful.
sales_data_aggregated.set_index('Date',inplace = True)

#Applying "seasonal_decompose" function to apply seasonal decomposition to sales of
#different components add together to create observed data. Based on output, additi
#apparent consistency in seasonal patterns, a well-defined trend, and that the resi
retail_decomp = seasonal_decompose(sales_data_aggregated['Sales'], model = 'additiv

#Creating new columns in "sales_data_aggregated" dataframe, with each corresponding
sales_data_aggregated['Trend'] = retail_decomp.trend
sales_data_aggregated['Seasonal'] = retail_decomp.seasonal
sales_data_aggregated['Residual'] = retail_decomp.resid

#Plotting the seasonal decomposition of the monthly retail data from 2014 to mid 20
decomp_plot = retail_decomp.plot()
decomp_plot.axes[0].set_title('')
decomp_plot.suptitle('Seasonal Decomposition of Average Monthly Retail Sales Across
plt.show()
```

Seasonal Decomposition of Average Monthly Retail Sales Across Businesses Over Time



A **seasonal decomposition** is a statistical technique used to break down time series data into its constituent components to better understand and analyze its underlying patterns.

There are 3 main components to a seasonal decomposition: The **trend component**, which depicts the general tendency of the data over the given time frame, the **seasonal component**, which shows the most fitting cyclical pattern that can be attributed to the data, and the **residual component**, which displays the random variability that remains after the trend and seasonal components are removed.

The trend and residual plots both indicate a **period of irregularity** in the data, specifically from March of 2020 to the summer of 2021. This abnormality can be attributed to the COVID-19 pandemic, which caused sales to **plunge drastically** in March of 2020, only to quickly recover and reach a period of **constant fluctuation** in the following months, before making **another abrupt spike** in the summer of 2021. This information is extremely valuable, as upon seeing it, I knew that a specific feature must be fed into my model to account for this time period.

The seasonal component plot is also insightful, showing that there are **peaks** (relative maxima) from around March to April and November to December of each year. It also reveals the presence of **relative minima** during February and October of each year. These findings seem reasonable based on real-life context, as consumers may feel pressured to spend more in the spring on summer essentials, and they also may spend more during the holiday season due to lower prices.

The seasonal plot indicates the need for **lag features**, which involve using a variable's value at a previous point as a feature in a model at the current time point.

As a result of the seasonal decomposition, I gained a better understanding of potential features that could enhance the overall performance of my model.

Descriptive Statistics

```
In [14]: pd.set_option('display.max_rows',None)
#print(sales_data_aggregated)
#print(retail_data)
#print(sales_data_aggregated['Sales'].head())
print(sales_data_aggregated['Sales'].describe())
#print(sales_data_aggregated['Sales'].nsmallest(10))

count      125.000000
mean     102228.655734
std      20266.706448
min      77696.236842
25%     84951.394737
50%     93614.783784
75%    123205.972222
max     137288.138889
Name: Sales, dtype: float64
```

Before beginning the model building process, I quickly examined the **descriptive statistics** of my target variable (Sales).

The minimum value of 77696.24 suggested a potential outlier that might need removal, although attempting to do so later unexpectedly decreased model performance.

Nevertheless, it is always wise to check the descriptive statistics of your data, as they can provide valuable information.

Review of RMSE (JUST FOR INFORMATIVE PURPOSES) IN THE CASE THAT READER IS UNFAMILIAR WITH THIS CONCEPT

To assess the performance of my Ridge Regression Model, I examine the **Root Mean Squared Error (RMSE)**, which refers to the square root of the average of the squared differences between the predicted and actual values.

Let's use a simple example to illustrate the concept of RMSE.

Assume we have both the actual and predicted average sales values across businesses from January to July 2024.

Actual values: [100,000 , 105,000 , 110,000 , 115,000 , 120,000]

Predicted Values: [102,000, 106,000, 108,000, 116,000, 119,000]

After calculating the differences (errors) between each corresponding actual and predicted value, we obtain [-2000, -1000, 2000, -1000, 1000].

We can take the square of the absolute value of each of these errors to get [4,000,000, 1,000,000, 4,000,000, 1,000,000, 1,000,000].

After computing the average of these squared errors, we obtain 2,200,000, which is the Mean Squared Error (MSE).

However, the MSE's value is in squared units, making it difficult to interpret directly in the context of the original data.

Therefore, we take the square root of our MSE to obtain 1483.24 as our RMSE.

This value tells us that, on average, the model's predictions would be off by about 1483.24 units in sales compared to the actual recorded sales values.

Since the average actual sales value in this example is 110,000, we could conclude that our predictions, on average, deviate from the actual sales values by about 1.3% of the mean sales value. This would indicate that our model is reasonably accurate.

Final Model

```
In [15]: from sklearn.metrics import mean_squared_error #Importing "mean_squared_error" function
from sklearn.linear_model import Ridge #Importing ridge regression algorithm to use
from sklearn.model_selection import TimeSeriesSplit, cross_val_score #Importing the

#Function that adds columns of feature variables to dataframe copy (to preserve original pd.to_datetime, by converting columns to datetime format, allows the usage of various
def retail_features(sales_data_aggregated):
    sales_data_aggregated = sales_data_aggregated.copy() #creating copy of df to prevent
    sales_data_aggregated['year_feature'] = sales_data_aggregated.index.year #capturing
    sales_data_aggregated['month_feature'] = sales_data_aggregated.index.month #capturing
    sales_data_aggregated['quarter_feature'] = sales_data_aggregated.index.quarter
    sales_data_aggregated['Lag_1'] = sales_data_aggregated['Sales'].shift(1) #capturing
    sales_data_aggregated['Lag_2'] = sales_data_aggregated['Sales'].shift(2) #capturing
    sales_data_aggregated['Lag_3'] = sales_data_aggregated['Sales'].shift(3) #capturing
    sales_data_aggregated['Rolling_Mean_3'] = sales_data_aggregated['Sales'].rolling(3).mean() #Creating feature to account for period of irregularity between March 2020 and
    sales_data_aggregated['COVID_Indicator'] = ((sales_data_aggregated.index >= '2020-03-01') * 1) #returning
    return sales_data_aggregated

# Applying the engineered features to the entire dataset
sales_data_aggregated_2 = retail_features(sales_data_aggregated)

# Handling NaN values with forward fill and bfill, which basically impute empty values
sales_data_aggregated_2.fillna(inplace=True)
sales_data_aggregated_2.bfill(inplace=True)

#Cannot use "train_test_split" because I am dealing with time series data here, so
```

```

#train_test_split randomly splits the data, breaking the temporal order and causing
#Instead, applying the "retail_features" function to the train and test subsets fro
train = sales_data_aggregated_2.loc[sales_data_aggregated_2.index < '2024-01'].copy
test = sales_data_aggregated_2.loc[sales_data_aggregated_2.index >= '2024-01'].copy

# Ensuring there are no NaN values in train and test sets
train.dropna(inplace=True)
test.dropna(inplace=True)

# Creating variables containing the features and the target variable
features = ['year_feature', 'month_feature', 'quarter_feature', 'Lag_1', 'Lag_2', 'COVID_Indicator']
target = 'Sales'

# Creating the train and test sets, with both containing the features and target
x_train = train[features]
y_train = train[target]
x_test = test[features]
y_test = test[target]

# Initializing the Ridge Regressor
retail_model = Ridge(alpha=1.0) # Regularization strength

# Performing 5-fold cross-validation using "TimeSeriesSplit" function
time_series_cv = TimeSeriesSplit(n_splits=5)
cv_scores = cross_val_score(retail_model, x_train, y_train, cv=time_series_cv, scoring='neg_mean_squared_error')
cv_scores = -cv_scores # Converting scores to positive values

# Printing performance metrics for cross validation
print(f'Cross-validated RMSE scores: {cv_scores}')
print(f'Mean cross-validated RMSE: {np.mean(cv_scores)}')

# Fitting the model on the entire training set
retail_model.fit(x_train, y_train)

# Making predictions on the test set and printing the results/performance metrics
test_predictions = retail_model.predict(x_test)
test_rmse = np.sqrt(mean_squared_error(y_test, test_predictions))

print(f'RMSE Score on Test set: {test_rmse:.2f}')

```

Cross-validated RMSE scores: [374.18895052 473.99775533 1525.73334322 473.968508
93.99435234]

Mean cross-validated RMSE: 588.3765818848949

RMSE Score on Test set: 25.16

To build my final model, I first created a function to create relevant features that could provide insight into my **target** variable, which is the Average Monthly Sales Across Businesses in America. Most of my features add value by accounting for trends in relation to specific time frames, such as each month or each fiscal quarter. One particularly robust feature is the **3-month rolling average**, which calculates the average values for the 3 months preceding a given month. I also included a "COVID_Indicator" feature, which instructed the model to also account for the specific trend that occurred during the aforementioned period of irregularity.

After updating my dataframe with the feature variables, I conducted a split of my entire dataset into training and testing sections, with the training section including all months before January 2024, and the testing section including all months after January 2024. I then assigned my feature and target variables to both these sets.

Once my training and testing sets were ready, I initialized a Ridge Regressor with an alpha value of 1, which means that **L2 Regularization** was applied to the model to **control overfitting** by penalizing large coefficients.

I finally performed 5-fold cross validation to check for overfitting, and then performed the standard feeding and predicting with my model to assess the results using RMSE.

```
In [16]: print(test['Sales'].describe())
```

```
count      5.000000
mean    136689.377778
std     652.845741
min    135675.000000
25%    136433.611111
50%    136909.833333
75%    137140.305556
max    137288.138889
Name: Sales, dtype: float64
```

Final Model Results and Algorithmic Comparison

The test set's RMSE score of **25.16** indicates that, on average, the Ridge Regression model's predictions for each month's average monthly sales only deviated from the actual values by about **.026%** of the mean sales value (136689.377778) for the testing data.

Therefore, this RMSE score reveals that the model is performing very well, as the sales values it predicts are very close to the true values, on average.

The average RMSE score across the 5 Cross-validation folds of **588.38** verifies that the model is not significantly overfitting onto the testing data, which means that the exquisite performance is not due to chance.

While deciding which algorithm to use for my model, I experimented with other options, such as the GradientBoostingRegressor and the RandomForestRegressor. However, the **relative simplicity** of the Ridge Regression algorithm ended up working better with my data, which highlights how the complex algorithm is not always the most appropriate one.

Forecasting into the Future

```
In [17]: #Calling "retail_features" function again and assigning to new dataframe because I
future_sales_data = retail_features(sales_data_aggregated)
```

```

#Applying backwards and forwards propagation again to take care of NaN values that
future_sales_data.ffill(inplace = True)
future_sales_data.bfill(inplace = True)
#Dropping all NaN values to make sure this issue is taken care of
future_sales_data.dropna(inplace = True)

#Now Using all data from Jan 2014 to May 2024 as training data, and assigning "feat
x_all = future_sales_data[features]

#Assigning the target variable, "Sales", as my y variable
y_all = future_sales_data[target]

#Initiating Ridge Regression Algorithm with alpha = 1.0 again
forecasting_model = Ridge(alpha = 1.0)

#Fitting Ridge Regression model again, but this time with all data from Jan 2014 to
forecasting_model.fit(x_all,y_all)

#Creating a new range of time from May 2024 to May 2024 at a 1 month frequency. Thi
future_months = pd.date_range('2024-06','2025-06',freq = '1ME')

#Creating new dataframe with index as the time range "future_months" that was just
future_months_df = pd.DataFrame(index = future_months)

#Creating a Boolean Mask, where the "True" values are based on the index of "future
#to query specifically the Sales data from this range when I need to predict for th
future_months_df['is_future'] = True

#Creating other end of Boolean Mask, where False values (ones not selected) correspo
#2024
future_sales_data['is_future'] = False

# Concatenating the dataframe "future_sales_data", which contains data from Jan 201
sales_into_future = pd.concat([future_sales_data,future_months_df])
#Applying the function sales_into_future dataframe to apply features to the part of
sales_into_future = retail_features(sales_into_future)

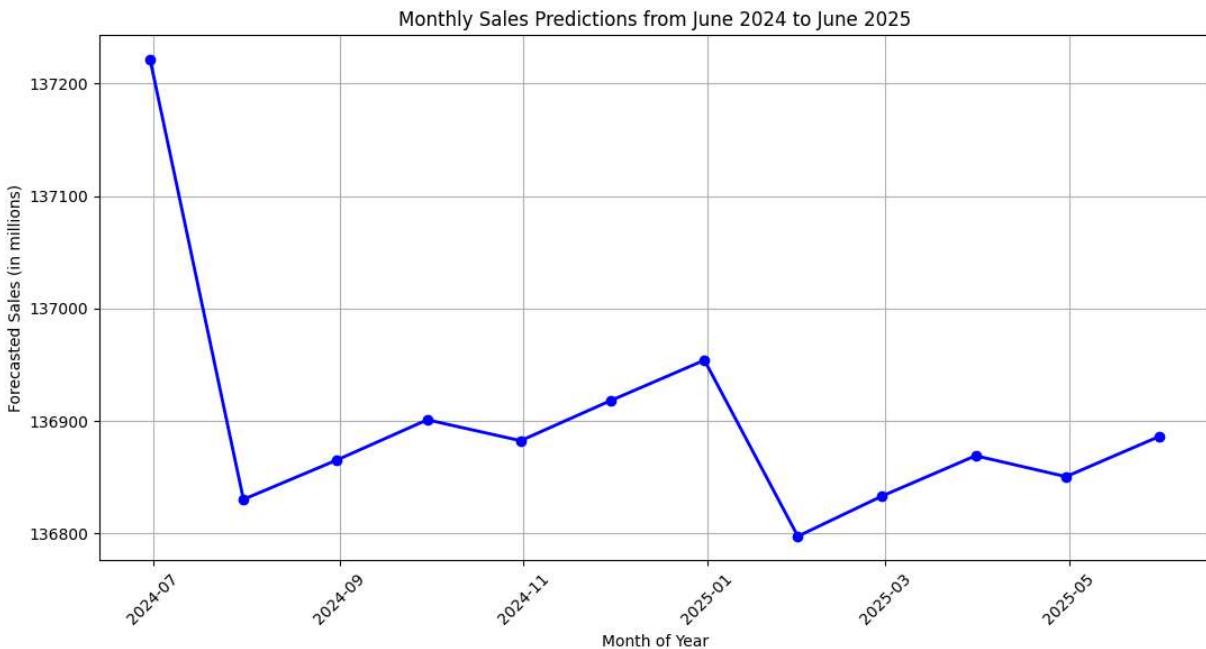
#Applying all the same cleaning techniques again since I had to apply the "retail_f
sales_into_future.ffill(inplace = True)
sales_into_future.bfill(inplace = True)
sales_into_future.dropna(inplace = True)

#Querying only data from May 2024 to May 2025 from the concatenated dataframe as I
future_data = sales_into_future.query('is_future').copy()
#Making predictions into future for the Monthly Average Sales
future_data['predictions'] = forecasting_model.predict(future_data[features])

#Adjusting a decently sized figure to display the trend of the predictions into the
plt.figure(figsize = (11,6))
#Plotting the months into future (index May 2024 to May 2025) vs the Predictions th
plt.plot(future_data.index, future_data['predictions'], marker = 'o', color = 'blue'
plt.title('Monthly Sales Predictions from June 2024 to June 2025') #Setting a title
plt.xlabel('Month of Year') #Setting the xlabel
plt.ylabel('Forecasted Sales (in millions)') #Setting the ylabel
plt.grid(True) #Setting grid for better visibility and doing other stuff for visibi

```

```
plt.xticks(rotation = 45)
plt.tight_layout()
plt.show() #Showing the plot
```



To forecast the sales values from June of 2024 to June of 2025, I essentially performed similar steps to those performed during the development of my final model in the previous section, but I needed to restart said process **to use all my data as training data**.

Therefore, I called my "retail_features" function again and assigned it to a new dataframe. I then applied similar NaN handling like before, and fit a new Ridge Regressor with data from Jan 2014 to May 2024.

However, I introduced the usage of Pandas' "**date_range**" function to create the index that was used for a new dataframe associated with the future months of May 2024 to May 2025. I also utilized a "boolean mask", where the indices of the dataframe for future months were set to "True". This allowed me to predict specifically for the future months later on.

I then **concatenated** the dataframes for the past months and the future months, before applying calling my "retail_features" function on this concatenated dataframe and repeating the corresponding code for cleaning.

I finally queried the range of future months using my Boolean Mask, and predicted with the model.

The resulting graph of the forecasted sales projects a sizeable dip from the end of June to August of 2024, followed by fluctuations for the remainder of the time frame. However, another smaller dip projected in January of 2025 results in a projection of the sales value being lower in June of 2025 than it was in June of 2024.

Regardless of whether this projection is reasonable or not, the power of programming for the purpose of Data Science can not be underestimated. With the appropriate features, model's like this one can be built and have the potential to make crucial predictions.

Conclusion

Over the course of this personal project, I have gained extensive knowledge of the various processes and techniques that were applied to achieve a well-performing model.

However, there are undoubtedly other ways in which my procedure could have been improved, such as by dealing with a larger set of data or forecasting into the future.

Nonetheless, I am glad I completed this project as it helped me get one step closer to my overall goal of becoming a capable Data Scientist.

Thank you for reading my report, and I hope you enjoyed it.