Sales_Department_Analysis

June 22, 2024

1 Optimizing Sales Performance: A Data-Driven Analysis of Retail Dynamics

1.0.1 Description

This project is a comprehensive data-driven analysis of the sales department's performance within a retail setting. The primary objective is to uncover critical insights that can drive strategic decisions to optimize sales and enhance overall business performance. The analysis employs a rich dataset that includes daily sales figures, customer counts, promotional activities, and competitive landscape details.

Key components of the project include:

1. **Data Cleaning and Preprocessing**:
   - Handling missing values and inconsistencies in the dataset to ensure accurate analysis.
   - Merging multiple data sources to create a cohesive dataset that includes store-specific details, such as competition proximity and promotional history.
2. **Exploratory Data Analysis (EDA)**:
   - Visualizing sales trends over time to identify seasonal patterns and peak sales periods.
   - Analyzing customer traffic to understand the relationship between customer counts and sales performance.
   - Assessing the impact of promotions and holidays on sales to determine the effectiveness of marketing strategies.
3. **Correlation and Causal Analysis**:
   - Investigating the correlation between various factors, such as promotional activities, competition distance, and sales performance.
   - Utilizing statistical methods to identify causal relationships that influence sales outcomes.
4. **Predictive Modeling**:
   - Developing predictive models to forecast future sales based on historical data and identified trends.
   - Implementing machine learning algorithms to enhance prediction accuracy and support decision-making processes.
5. **Competitive Analysis**:
   - Analyzing the impact of competitor proximity and store characteristics on sales performance.
   - Evaluating how different store types and assortments affect customer preferences and sales results.
6. **Actionable Insights and Recommendations**:

- Summarizing findings to provide actionable insights that can inform sales strategies and promotional planning.
- Offering recommendations to optimize inventory management, promotional activities, and customer engagement based on data-driven evidence.

The outcome of this project is a detailed report that presents a thorough understanding of the factors driving sales performance in the retail environment. The insights gained from this analysis will empower the sales department to make informed decisions, enhance marketing strategies, and ultimately boost sales and customer satisfaction.

Laying the Foundation for Data Discovery

To embark on our data exploration journey, we're arming ourselves with these essential tools:

1. **NumPy (as np):** This is our numerical powerhouse. It empowers us to perform calculations on data that's organized in grids or tables, making it the backbone of our quantitative analysis.

2. **Pandas (as pd):** Think of Pandas as our data concierge. It helps us effortlessly wrangle and manage our data, whether it's cleaning up messy information, merging different datasets, or extracting the precise insights we need.

3. **Visualization Libraries:** Because a picture is worth a thousand words (and a thousand data points!), we're using:

    - **Matplotlib (as plt):** Our trusted tool for crafting clear, informative graphs and charts that help us visually grasp patterns and trends within our data.
    - **Plotly Express (as px):** This takes our visualizations to the next level, adding interactivity and dynamism that allow us to dive deeper into the nuances of our data.
    - **Plotly Figure Factory (as ff):** This provides us with ready-made templates for common statistical visualizations, saving us time and effort while ensuring our visuals are polished and professional.
    - **Seaborn (as sns):** This library enhances our visual storytelling capabilities, offering specialized statistical plots that are both aesthetically pleasing and insightful.

4. **Additional Tools:**

    - **Datetime:** This enables us to handle dates and times effectively, ensuring we can analyze any temporal aspects of our data, such as trends over time or seasonal variations.
    - **Warnings Filter:** This helps us keep our focus on the big picture by temporarily suppressing warning messages that might otherwise clutter our output.

The Takeaway

With this carefully curated toolkit, we're poised to unlock the secrets hidden within our data. Whether we're identifying outliers, revealing correlations, or uncovering unexpected patterns, these tools will empower us to extract meaningful knowledge and make data-driven decisions with confidence.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
import plotly.express as px
import plotly.figure_factory as ff
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
```

- This line of code reads a CSV file named "train.csv" into a Pandas DataFrame named "train." This DataFrame will hold the data for analysis and modeling.

```
[2]: train = pd.read_csv('train.csv')
```

- This code displays the first 5 rows of the DataFrame `train`, providing a quick overview of its structure, column names, and the initial values in each column.

```
[3]: train.head()
```

```
[3]:    Store  DayOfWeek        Date  Sales  Customers  Open  Promo StateHoliday  \
     0      1          5  2015-07-31   5263        555     1      1            0
     1      2          5  2015-07-31   6064        625     1      1            0
     2      3          5  2015-07-31   8314        821     1      1            0
     3      4          5  2015-07-31  13995       1498     1      1            0
     4      5          5  2015-07-31   4822        559     1      1            0

        SchoolHoliday
     0              1
     1              1
     2              1
     3              1
     4              1
```

- This code will return the dimensions of the DataFrame `train` as a tuple (number of rows, number of columns).

```
[4]: train.shape
```

```
[4]: (1017209, 9)
```

- *This code provides a concise summary of the DataFrame* ***`train`****. It will display:*
- The column names.
- The data types of each column (e.g., int64, float64, object).
- The number of non-null values in each column.
- The memory usage of the DataFrame.

```
[5]: train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1017209 entries, 0 to 1017208
Data columns (total 9 columns):
 #   Column         Non-Null Count    Dtype 
---  ------         --------------    ----- 
 0   Store          1017209 non-null  int64 
 1   DayOfWeek      1017209 non-null  int64 
 2   Date           1017209 non-null  object
 3   Sales          1017209 non-null  int64 
 4   Customers      1017209 non-null  int64 
 5   Open           1017209 non-null  int64 
 6   Promo          1017209 non-null  int64 
 7   StateHoliday   1017209 non-null  object
 8   SchoolHoliday  1017209 non-null  int64 
dtypes: int64(7), object(2)
memory usage: 69.8+ MB
```

- The code `train.describe().T` calculates summary statistics for the numerical columns in the DataFrame `train`. The `.T` at the end transposes the results, making them easier to read by switching rows and columns.

```
[6]: train.describe().T
```

```
[6]:                   count         mean          std  min     25%     50%  \
     Store          1017209.0   558.429727   321.908651  1.0   280.0   558.0   
     DayOfWeek      1017209.0     3.998341     1.997391  1.0     2.0     4.0   
     Sales          1017209.0  5773.818972  3849.926175  0.0  3727.0  5744.0   
     Customers      1017209.0   633.145946   464.411734  0.0   405.0   609.0   
     Open           1017209.0     0.830107     0.375539  0.0     1.0     1.0   
     Promo          1017209.0     0.381515     0.485759  0.0     0.0     0.0   
     SchoolHoliday  1017209.0     0.178647     0.383056  0.0     0.0     0.0   

                       75%      max  
     Store           838.0   1115.0  
     DayOfWeek         6.0      7.0  
     Sales          7856.0  41551.0  
     Customers       837.0   7388.0  
     Open              1.0      1.0  
     Promo             1.0      1.0  
     SchoolHoliday     0.0      1.0  
```

- **Sales figures vary significantly across stores, with the average around 5774 euros and a notable outlier reaching 41551 euros.**

-

1.1 Customer traffic also varies widely among stores, with an average of 633 customers per store and a peak of 7388 at one location.

- This code reads data from a CSV file named "store.csv" and stores it into a Pandas DataFrame named `store_data`. This DataFrame will be used for further analysis or processing.

[7]:

```python
store_data = pd.read_csv('store.csv')
```

- This code displays the first 5 rows of the DataFrame `store_data`, providing a quick glimpse of its structure, column names, and the initial values in each column.

[8]:

```python
store_data.head()
```

[8]:

```
   Store StoreType Assortment  CompetitionDistance  CompetitionOpenSinceMonth  \
0      1         c          a               1270.0                        9.0
1      2         a          a                570.0                       11.0
2      3         a          a              14130.0                       12.0
3      4         c          c                620.0                        9.0
4      5         a          a              29910.0                        4.0

   CompetitionOpenSinceYear  Promo2  Promo2SinceWeek  Promo2SinceYear  \
0                    2008.0       0              NaN              NaN
1                    2007.0       1             13.0           2010.0
2                    2006.0       1             14.0           2011.0
3                    2009.0       0              NaN              NaN
4                    2015.0       0              NaN              NaN

     PromoInterval
0              NaN
1  Jan,Apr,Jul,Oct
2  Jan,Apr,Jul,Oct
3              NaN
4              NaN
```

[9]:

```python
store_data.shape
```

[9]:

```
(1115, 10)
```

[10]:

```python
store_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1115 entries, 0 to 1114
Data columns (total 10 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Store                      1115 non-null   int64
 1   StoreType                  1115 non-null   object
 2   Assortment                 1115 non-null   object
 3   CompetitionDistance        1112 non-null   float64
 4   CompetitionOpenSinceMonth  761 non-null    float64
 5   CompetitionOpenSinceYear   761 non-null    float64
 6   Promo2                     1115 non-null   int64
 7   Promo2SinceWeek            571 non-null    float64
 8   Promo2SinceYear            571 non-null    float64
 9   PromoInterval              571 non-null    object
dtypes: float64(5), int64(2), object(3)
memory usage: 87.2+ KB
```

[11]:

```
store_data.describe().T
```

[11]:

```
                           count         mean          std     min     25%  \
Store                     1115.0   558.000000   322.017080     1.0   279.5
CompetitionDistance       1112.0  5404.901079  7663.174720    20.0   717.5
CompetitionOpenSinceMonth  761.0     7.224704     3.212348     1.0     4.0
CompetitionOpenSinceYear   761.0  2008.668857     6.195983  1900.0  2006.0
Promo2                    1115.0     0.512108     0.500078     0.0     0.0
Promo2SinceWeek            571.0    23.595447    14.141984     1.0    13.0
Promo2SinceYear            571.0  2011.763573     1.674935  2009.0  2011.0

                             50%     75%      max
Store                      558.0   836.5   1115.0
CompetitionDistance       2325.0  6882.5  75860.0
CompetitionOpenSinceMonth    8.0    10.0     12.0
CompetitionOpenSinceYear  2010.0  2013.0   2015.0
Promo2                       1.0     1.0      1.0
Promo2SinceWeek             22.0    37.0     50.0
Promo2SinceYear           2012.0  2013.0   2015.0
```

-

1.2 The proximity of competitors to each store also varies significantly, with an average distance of 5405 meters and one store located as far as 75860 meters from its nearest competitor.

[12]:

```
train.isnull().sum()
```

[12]:

```
Store            0
DayOfWeek        0
Date             0
Sales            0
Customers        0
Open             0
Promo            0
StateHoliday     0
SchoolHoliday    0
dtype: int64
```

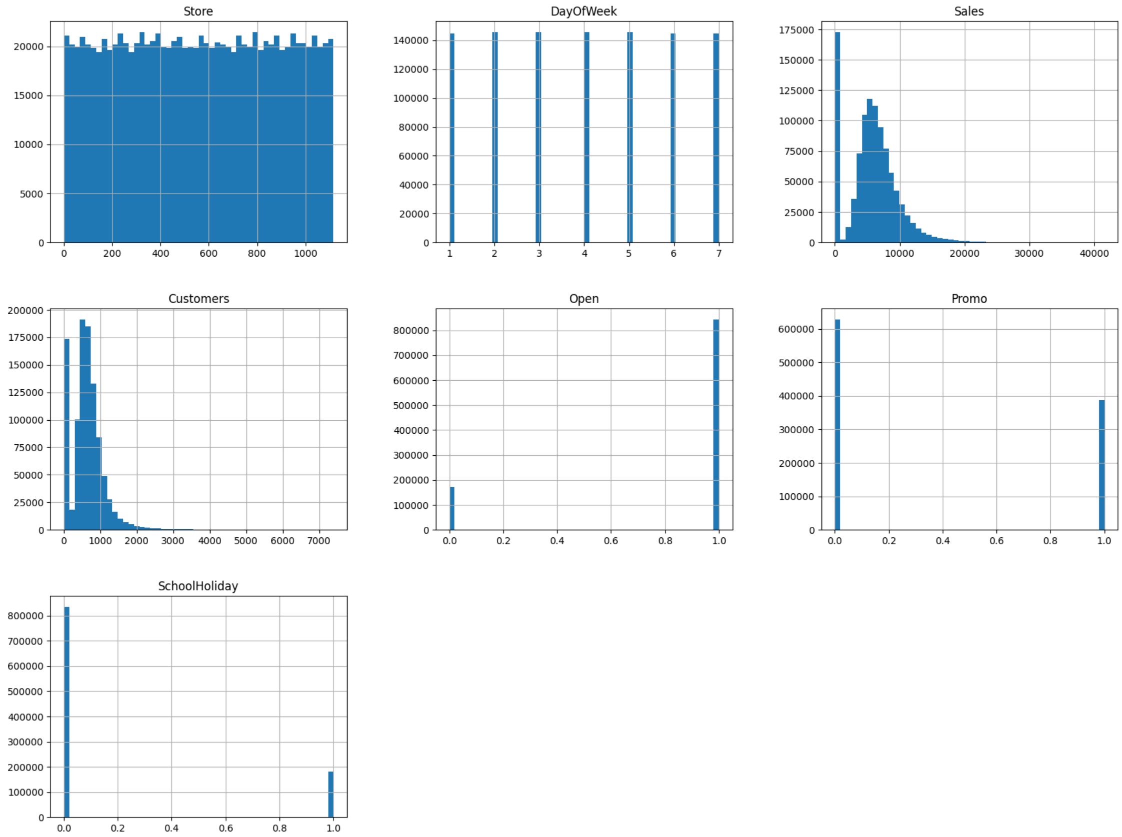
- **It’s evident that the train dataset contains no null values.**

This code generates histograms for the numerical columns in the DataFrame `train`.

- `bins=50`: Specifies that each histogram will have 50 bins (intervals) to group the data values.

- `figsize=(20,15)`: Sets the size of the entire figure containing all the histograms to 20 inches wide and 15 inches tall.
- `plt.show()`: Displays the figure with all the generated histograms.

[13]:
```
train.hist(bins=50, figsize=(20,15))
plt.show()
```



-

1.3 Here we can observe that sales are evenly distributed throughout the days of the week.

This code counts the occurrences of each unique value in the ‘Open’ column of the DataFrame **`train`**. The ‘Open’ column indicates whether a store was open or closed on a particular day, this code will tell us how many instances there are of the store being open or closed in the dataset.

[14]:
```
train['Open'].value_counts()
```

[14]:
```
Open
1    844392
0    172817
Name: count, dtype: int64
```

- **The data reveals a significant difference between the number of open stores (844,392) and closed stores (172,817), indicating a substantial active retail landscape.**

-

1.4 Considering the substantial number of closed stores with missing or zero values, it would be beneficial to filter out this data and focus our analysis on active stores.

- This code filters the DataFrame `train` to keep only the rows where the value in the column named `'Open'` is equal to 1. The resulting filtered DataFrame is then reassigned to the variable `train`, effectively overwriting the original DataFrame.

```
[15]: train = train[train['Open'] == 1]
```

```
[16]: train
```

```
[16]:          Store  DayOfWeek        Date  Sales  Customers  Open  Promo  \
0            1          5  2015-07-31   5263        555     1      1
1            2          5  2015-07-31   6064        625     1      1
2            3          5  2015-07-31   8314        821     1      1
3            4          5  2015-07-31  13995       1498     1      1
4            5          5  2015-07-31   4822        559     1      1
…          …          …           …     …          …    …      …
1016776    682          2  2013-01-01   3375        566     1      0
1016827    733          2  2013-01-01  10765       2377     1      0
1016863    769          2  2013-01-01   5035       1248     1      0
1017042    948          2  2013-01-01   4491       1039     1      0
1017190   1097          2  2013-01-01   5961       1405     1      0

        StateHoliday  SchoolHoliday
0                  0              1
1                  0              1
2                  0              1
3                  0              1
4                  0              1
…                …              …
1016776            a              1
1016827            a              1
1016863            a              1
1017042            a              1
1017190            a              1

[844392 rows x 9 columns]
```

- **Since the remaining data exclusively represents open stores, we can proceed to remove the "open" column as it no longer provides any differentiating information.**

[17]:
```
train.drop('Open', axis=1, inplace=True)
```

[18]:
```
train
```

[18]:
```
         Store  DayOfWeek        Date  Sales  Customers  Promo StateHoliday  \
0            1          5  2015-07-31   5263        555      1            0
1            2          5  2015-07-31   6064        625      1            0
2            3          5  2015-07-31   8314        821      1            0
3            4          5  2015-07-31  13995       1498      1            0
4            5          5  2015-07-31   4822        559      1            0
…        …        …         …    …        …    …          …
1016776    682          2  2013-01-01   3375        566      0            a
1016827    733          2  2013-01-01  10765       2377      0            a
1016863    769          2  2013-01-01   5035       1248      0            a
1017042    948          2  2013-01-01   4491       1039      0            a
1017190   1097          2  2013-01-01   5961       1405      0            a

         SchoolHoliday
0                    1
1                    1
2                    1
3                    1
4                    1
…                …
1016776              1
1016827              1
1016863              1
1017042              1
1017190              1

[844392 rows x 8 columns]
```

[19]:
```
train.describe().T
```

[19]:
```
                  count         mean          std  min     25%     50%  \
Store          844392.0   558.422920   321.731914  1.0   280.0   558.0
DayOfWeek      844392.0     3.520361     1.723689  1.0     2.0     3.0
Sales          844392.0  6955.514291  3104.214680  0.0  4859.0  6369.0
Customers      844392.0   762.728395   401.227674  0.0   519.0   676.0
Promo          844392.0     0.446352     0.497114  0.0     0.0     0.0
SchoolHoliday  844392.0     0.193580     0.395103  0.0     0.0     0.0

                  75%      max
Store           837.0   1115.0
DayOfWeek         5.0      7.0
Sales          8360.0  41551.0
Customers       893.0   7388.0
```

```
Promo              1.0       1.0
SchoolHoliday      0.0       1.0
```

- **Following the removal of closed stores, the average sales per store significantly increased to 6955 euros, and the average number of customers per store also rose to 762, reflecting the performance of actively operating stores.**

[20]:
```
store_data.isnull().sum()
```

[20]:
```
Store                          0
StoreType                      0
Assortment                     0
CompetitionDistance            3
CompetitionOpenSinceMonth    354
CompetitionOpenSinceYear     354
Promo2                         0
Promo2SinceWeek              544
Promo2SinceYear              544
PromoInterval                544
dtype: int64
```

The following columns in the `store_data` DataFrame have missing values:

- `CompetitionDistance`: 3 missing values
- `CompetitionOpenSinceMonth`: 354 missing values
- `CompetitionOpenSinceYear`: 354 missing values
- `Promo2SinceWeek`: 544 missing values
- `Promo2SinceYear`: 544 missing values
-

1.5 `PromoInterval`: 544 missing values

- This code filters the DataFrame `store_data` to select only the rows where the values in the column `'Promo2SinceWeek'` are missing (NaN). This is often used to identify and further investigate the rows where this specific piece of information is not available.

[21]:
```
store_data[store_data['Promo2SinceWeek'].isna()]
```

[21]:
```
      Store StoreType Assortment  CompetitionDistance  \
0         1         c          a               1270.0
3         4         c          c                620.0
4         5         a          a              29910.0
5         6         a          a                310.0
6         7         a          c              24000.0
…     …       …        …                  …
1107   1108         a          a                540.0
```

```
1109   1110         c          c                900.0
1111   1112         c          c               1880.0
1112   1113         a          c               9260.0
1113   1114         a          c                870.0

      CompetitionOpenSinceMonth  CompetitionOpenSinceYear  Promo2  \
0                           9.0                    2008.0       0
3                           9.0                    2009.0       0
4                           4.0                    2015.0       0
5                          12.0                    2013.0       0
6                           4.0                    2013.0       0
…                         …                       …     …
1107                        4.0                    2004.0       0
1109                        9.0                    2010.0       0
1111                        4.0                    2006.0       0
1112                        NaN                       NaN       0
1113                        NaN                       NaN       0

      Promo2SinceWeek  Promo2SinceYear PromoInterval
0                 NaN              NaN           NaN
3                 NaN              NaN           NaN
4                 NaN              NaN           NaN
5                 NaN              NaN           NaN
6                 NaN              NaN           NaN
…               …              …           …
1107              NaN              NaN           NaN
1109              NaN              NaN           NaN
1111              NaN              NaN           NaN
1112              NaN              NaN           NaN
1113              NaN              NaN           NaN

[544 rows x 10 columns]
```

- ### 1.6 Given that `Promo2SinceWeek`, `Promo2SinceYear`, and `PromoInterval` are NaN whenever `Promo2` is 0, it's reasonable to assume these missing values indicate no participation in the second promotion. Therefore, we can safely replace these NaN values with 0 to represent this absence of promotional activity.

- These lines of code fill the missing values (NaN) in multiple columns of the DataFrame `store_data` with the value 0. The `inplace=True` argument means the operation is performed directly on the original DataFrame, modifying it without creating a new copy.

The specific columns being modified are:

- `Promo2SinceWeek`
- `Promo2SinceYear`

- `PromoInterval`
- `CompetitionOpenSinceMonth`
- `CompetitionOpenSinceYear`

```
[22]: store_data['Promo2SinceWeek'].fillna(0, inplace=True)
      store_data['Promo2SinceYear'].fillna(0, inplace=True)
      store_data['PromoInterval'].fillna(0, inplace=True)
      store_data['CompetitionOpenSinceMonth'].fillna(0, inplace=True)
      store_data['CompetitionOpenSinceYear'].fillna(0, inplace=True)
```

- **To address the missing values in `CompetitionDistance`, we can impute them with the mean value calculated from the existing non-null entries in this column.**

```
[23]: store_data['CompetitionDistance'].fillna(store_data['CompetitionDistance'].
        ↪mean(), inplace=True)
```

- This code merges the DataFrames `train` and `store_data` into a new DataFrame named `data` based on a common column called `'Store'`. This means it combines rows from both DataFrames where the values in the `'Store'` column match.
- `pd.merge()`: This is the Pandas function used for merging DataFrames.
- `how='inner'`: Specifies an inner join, meaning only rows with matching values in the `'Store'` column from both DataFrames will be included in the result.
- `on='Store'`: Indicates that the merging should be done based on the values in the column named `'Store'`.

```
[24]: data = pd.merge(train, store_data, how = 'inner', on='Store')
```

- This code transposes the first 5 rows of the DataFrame `data` and displays the result. Transposing means switching the rows and columns, which can make it easier to view and compare the values in each column.

```
[25]: data.head().T
```

```
[25]:                                     0           1           2           3  \
      Store                               1           1           1           1
      DayOfWeek                           5           4           3           2
      Date                       2015-07-31  2015-07-30  2015-07-29  2015-07-28
      Sales                            5263        5020        4782        5011
      Customers                         555         546         523         560
      Promo                               1           1           1           1
      StateHoliday                        0           0           0           0
      SchoolHoliday                       1           1           1           1
      StoreType                           c           c           c           c
      Assortment                          a           a           a           a
      CompetitionDistance            1270.0      1270.0      1270.0      1270.0
      CompetitionOpenSinceMonth         9.0         9.0         9.0         9.0
      CompetitionOpenSinceYear       2008.0      2008.0      2008.0      2008.0
```

```
Promo2                               0            0            0            0
Promo2SinceWeek                    0.0          0.0          0.0          0.0
Promo2SinceYear                    0.0          0.0          0.0          0.0
PromoInterval                        0            0            0            0

                                     4
Store                                1
DayOfWeek                            1
Date                        2015-07-27
Sales                             6102
Customers                          612
Promo                                1
StateHoliday                         0
SchoolHoliday                        1
StoreType                            c
Assortment                           a
CompetitionDistance             1270.0
CompetitionOpenSinceMonth          9.0
CompetitionOpenSinceYear        2008.0
Promo2                               0
Promo2SinceWeek                    0.0
Promo2SinceYear                    0.0
PromoInterval                        0
```

This code does the following:

1. **Calculates Correlation:** It computes the correlation between the `Sales` column and all other numerical columns in the DataFrame `data`. The results are sorted in descending order.
2. **Creates Bar Plot:** It creates an interactive bar plot using Plotly Express (`px.bar`) to visualize the calculated correlations.
   - The x-axis displays the feature names (`sales_corr.index`).
   - The y-axis shows the correlation values (`sales_corr.values`).
3. **Enhances Plot:** It adjusts the plot's layout for better readability:
   - Sets clear labels for the x-axis ('Features') and y-axis ('Correlation with Sales').
   - Adds a title to the plot ('Correlation of Features with Sales').
   - Rotates the x-axis labels by 45 degrees for better spacing and readability.
4. **Displays Plot:** It shows the final interactive plot, allowing you to explore and interact with the visualization.

[26]:

```python
sales_corr = data.corr(numeric_only=True)['Sales'].sort_values(ascending=False)

# Create a bar plot using Plotly
fig = px.bar(
    x=sales_corr.index,
    y=sales_corr.values,
    labels={'x': 'Features', 'y': 'Correlation with Sales'},
    title='Correlation of Features with Sales'
)

# Update layout for better visualization
fig.update_layout(
    xaxis_title='Features',
    yaxis_title='Correlation with Sales',
    xaxis_tickangle=-45
)

fig.show()
```

This code calculates and visualizes a correlation matrix heatmap for numerical columns in the DataFrame `data`.

1. **Calculates Correlation Matrix:** It computes pairwise correlations between all numerical columns in `data`. The results are rounded to 5 decimal places for clarity.
2. **Creates Heatmap:** It uses Plotly Figure Factory (`ff.create_annotated_heatmap`) to create a heatmap:
    - `z`: The correlation matrix values are used as the color intensity for each cell in the heatmap.
    - `x` and `y`: The column names are used as labels for the x and y axes of the heatmap.
    - `colorscale='Viridis'`: A color scheme is applied to visually represent the correlation values.
    - `showscale=True`: A color bar is added to the side to interpret the correlation values.
3. **Enhances Heatmap:** The layout is adjusted for improved readability:
    - Adds a title "Correlation Matrix Heatmap".
    - Labels the x and y axes as "Features".
    - Sets the width and height of the plot for optimal viewing.
4. **Displays Heatmap:** The final heatmap is displayed, allowing for easy identification of relationships between different features.

[27]:

```
# Calculate the correlation matrix
corr_matrix = data.corr(numeric_only=True)
corr_matrix_rounded = np.round(corr_matrix, decimals=5)

# Create a heatmap using Plotly
fig = ff.create_annotated_heatmap(
    z=corr_matrix_rounded.values,
    x=corr_matrix_rounded.columns.tolist(),
    y=corr_matrix_rounded.columns.tolist(),
    colorscale='Viridis',
    showscale=True
)
```

```
# Update layout for better visualization
fig.update_layout(
    title='Correlation Matrix Heatmap',
    xaxis_title='Features',
    yaxis_title='Features',
    width=1000,
    height=800
)

fig.show()
```

This code extracts year, month, and day components from the `Date` column in the DataFrame `data` and stores them as separate columns: `year`, `month`, and `day` respectively. This is often done to make it easier to analyze or visualize data based on specific time periods.

- `pd.DatetimeIndex`: This creates an index of datetime objects from the `Date` column, making it easier to work with time-based operations.
- `.year`, `.month`, `.day`: These attributes are used to extract the respective components from each datetime object in the index.

[28]:

```
data['year'] = pd.DatetimeIndex(data['Date']).year
data['month'] = pd.DatetimeIndex(data['Date']).month
data['day'] = pd.DatetimeIndex(data['Date']).day
```

[29]:

```
data.head()
```

[29]:

```
   Store  DayOfWeek        Date  Sales  Customers  Promo StateHoliday  \
0      1          5  2015-07-31   5263        555      1            0
1      1          4  2015-07-30   5020        546      1            0
2      1          3  2015-07-29   4782        523      1            0
3      1          2  2015-07-28   5011        560      1            0
4      1          1  2015-07-27   6102        612      1            0

   SchoolHoliday StoreType Assortment  CompetitionDistance  \
0              1         c          a               1270.0
1              1         c          a               1270.0
2              1         c          a               1270.0
3              1         c          a               1270.0
4              1         c          a               1270.0

   CompetitionOpenSinceMonth  CompetitionOpenSinceYear  Promo2  \
0                        9.0                    2008.0       0
1                        9.0                    2008.0       0
2                        9.0                    2008.0       0
3                        9.0                    2008.0       0
4                        9.0                    2008.0       0
```

```
   Promo2SinceWeek  Promo2SinceYear PromoInterval  year  month  day
0              0.0              0.0             0  2015      7   31
1              0.0              0.0             0  2015      7   30
2              0.0              0.0             0  2015      7   29
3              0.0              0.0             0  2015      7   28
4              0.0              0.0             0  2015      7   27
```

This code calculates and visualizes the average monthly sales.

1. **Calculates Monthly Average Sales:**
   - It groups the data by `month`.
   - Calculates the mean `Sales` for each month.
   - Resets the index to make `month` a regular column for easier plotting.
2. **Creates Line Plot:**
   - It uses Plotly Express (`px.line`) to create an interactive line plot.
   - The x-axis represents the months.
   - The y-axis represents the average sales.
   - Markers are added to each data point for clarity.
   - The title of the plot is set to "Average Sales Volume per Month".
3. **Displays Plot:**
   - The interactive line plot is shown, allowing you to explore the monthly sales trends visually.

[30]:
```
monthly_mean_sales = data.groupby('month')['Sales'].mean().reset_index()

# Plot the data using Plotly
fig = px.line(monthly_mean_sales, x='month', y='Sales', title='Average Sales␣
 ↪Volume per Month', markers=True)
fig.show()
```

-

1.7 Sales data reveals a peak average during the Christmas season

This code calculates and visualizes the average number of customers per month.

1. **Calculates Monthly Mean Customers:**
   - Groups the DataFrame `data` by `month`.
   - Calculates the mean number of `Customers` for each month.
   - Resets the index to have `month` as a column for plotting.
2. **Creates Line Plot:**
   - Uses Plotly Express (`px.line`) to generate an interactive line plot.
   - The x-axis represents the months (`month`).
   - The y-axis shows the average number of customers (`Customers`).
   - Adds markers to data points for better visualization.
   - Sets the title as "Mean Number of Customers per Month".
3. **Displays Plot:**
   - Renders the line plot, allowing for interactive exploration of the monthly customer trends.

[31]:

```
monthly_mean_customers = data.groupby('month')['Customers'].mean().reset_index()


# Plot the data using Plotly
fig = px.line(monthly_mean_customers, x='month', y='Customers', title='Mean␣
 ↪Number of Customers per Month', markers=True)
fig.show()
```

-

1.8 Mirroring the sales data, customer traffic also peaks during the Christmas season.

This code calculates and visualizes the average number of customers per day.

1. **Calculates Daily Mean Customers:**
   - It groups the DataFrame `data` by the `day` column.
   - Calculates the mean number of `Customers` for each day.
   - Resets the index to have `day` as a regular column for easy plotting.
2. **Creates Line Plot:**
   - It uses Plotly Express (`px.line`) to generate an interactive line plot.
   - The x-axis represents the days of the month (`day`).
   - The y-axis represents the average number of customers (`Customers`).
   - It adds markers to each data point for clarity.
   - Sets the title as "Mean Number of Customers per Day".
3. **Displays Plot:**
   - It renders the interactive line plot, enabling you to explore how the average number of customers varies throughout the days of the month.

[32]:

```
daily_mean_customers = data.groupby('day')['Customers'].mean().reset_index()

# Plot the data using Plotly
fig = px.line(daily_mean_customers, x='day', y='Customers', title='Mean Number␣
 ↪of Customers per Day',markers=True)
fig.show()
```

This code calculates and visualizes the average daily sales.

1. **Calculates Daily Average Sales:**
   - It groups the data by `day`.
   - Computes the mean `Sales` for each day of the month.
   - Resets the index to make `day` a regular column for easy plotting.
2. **Creates Line Plot:**
   - Utilizes Plotly Express (`px.line`) to create an interactive line plot.
   - Sets the x-axis to represent the days of the month (`day`).
   - Sets the y-axis to represent the average sales (`Sales`).
   - Adds markers to each data point for visual clarity.
   - Titles the plot "Average Sales Volume per Day".
3. **Displays Plot:**

    - Shows the interactive line plot, enabling exploration of daily sales trends across the month.

[33]:
```
daily_mean_sales = data.groupby('day')['Sales'].mean().reset_index()

# Plot the data using Plotly
fig = px.line(daily_mean_sales, x='day', y='Sales', title='Average Sales Volume␣
 ↪per Day',markers=True)
fig.show()
```

This code calculates and visualizes the average sales volume for each day of the week.

1. **Calculates Mean Sales by Day of Week:**
    - Groups the `data` DataFrame by the `DayOfWeek` column.
    - Calculates the mean `Sales` for each day of the week.
    - Resets the index, creating a new DataFrame with columns `DayOfWeek` and the corresponding average `Sales`.
2. **Creates Line Plot:**
    - Utilizes Plotly Express (`px.line`) to create an interactive line plot.
    - The x-axis displays the days of the week (`DayOfWeek`).
    - The y-axis shows the average sales for each day (`Sales`).
    - Adds markers to each data point for visual emphasis.
    - Sets the title of the plot to "Average Sales Volume per Day Of Week".
3. **Displays Plot:**
    - Shows the interactive line plot, enabling you to observe the weekly sales patterns.

[34]:
```
week_mean_sales = data.groupby('DayOfWeek')['Sales'].mean().reset_index()

# Plot the data using Plotly
fig = px.line(week_mean_sales, x='DayOfWeek', y='Sales', title='Average Sales␣
 ↪Volume per Day Of Week',markers=True)
fig.show()
```

-

1.9 The highest sales performance was consistently observed on Sundays and Mondays.

This code calculates the average sales per store type and date, sorts the results by sales, and then creates a line plot to visualize the trends.

1. **Calculate Mean Sales:**
    - Groups the DataFrame `data` by both `StoreType` and `Date`.
    - Calculates the mean `Sales` for each combination of store type and date.
    - Resets the index to have `StoreType`, `Date`, and the corresponding average `Sales` as separate columns.
2. **Sort Data:**
    - Sorts the resulting DataFrame `mean_sales` by the `Sales` column in descending order, so the highest average sales are displayed first.

3. **Create Plot:**
   - Uses Plotly Express (`px.line`) to create an interactive line plot.
   - The x-axis represents the dates (`Date`).
   - The y-axis represents the average sales (`Sales`).
   - Each line on the plot corresponds to a different store type, indicated by the color (`color='StoreType'`).
   - The title of the plot is set to "Mean Sales per Store Type and Date".
4. **Display Plot:**
   - Renders the interactive line plot, allowing you to compare the average sales trends for different store types over time.

[35]:

```
mean_sales = data.groupby(['StoreType', 'Date'])['Sales'].mean().reset_index()

# Sort the data by sales in descending order
sorted_mean_sales = mean_sales.sort_values(by='Sales', ascending=False)

# Plot the data using Plotly
fig = px.line(sorted_mean_sales, x='Date', y='Sales', color='StoreType',␣
 ↪title='Mean Sales per Store Type and Date')
fig.show()
```

This code calculates and visualizes the average sales for each promotional status (Promo).

1. **Calculates Mean Sales by Promo:**
   - Groups the DataFrame `data` by the `Promo` column.
   - Computes the mean `Sales` for each distinct value in `Promo` (e.g., 0 for no promotion, 1 for promotion).
   - Resets the index to create a new DataFrame with columns `Promo` and the corresponding average `Sales`.
2. **Creates Bar Plot:**
   - Utilizes Plotly Express (`px.bar`) to generate an interactive bar chart.
   - The x-axis represents the different promotional statuses (`Promo`).
   - The y-axis represents the average sales (`Sales`) for each status.
   - Sets the title of the plot as "Mean Sales by Promotional Status".
3. **Displays Plot:**
   - Shows the interactive bar chart, allowing for a visual comparison of average sales under different promotional conditions.

[36]:

```
promo_sales = data.groupby('Promo')['Sales'].mean().reset_index()
fig = px.bar(promo_sales, x='Promo', y='Sales', title='Mean Sales by␣
 ↪Promotional Status')
fig.show()
```

- **Promotional efforts appear to have a positive impact on sales figures.**

2 Model Training

- **This code installs the Prophet library, a time series forecasting tool developed by Facebook. It's designed to be user-friendly and adaptable to various forecasting scenarios. The exclamation mark at the beginning indicates that this is a command to be executed in a terminal or command prompt environment.**

[37]:
```
!pip install prophet
```

```
Requirement already satisfied: prophet in /usr/local/lib/python3.10/dist-
packages (1.1.5)
Requirement already satisfied: cmdstanpy>=1.0.4 in
/usr/local/lib/python3.10/dist-packages (from prophet) (1.2.4)
Requirement already satisfied: numpy>=1.15.4 in /usr/local/lib/python3.10/dist-
packages (from prophet) (1.25.2)
Requirement already satisfied: matplotlib>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from prophet) (3.7.1)
Requirement already satisfied: pandas>=1.0.4 in /usr/local/lib/python3.10/dist-
packages (from prophet) (2.0.3)
Requirement already satisfied: holidays>=0.25 in /usr/local/lib/python3.10/dist-
packages (from prophet) (0.51)
Requirement already satisfied: tqdm>=4.36.1 in /usr/local/lib/python3.10/dist-
packages (from prophet) (4.66.4)
Requirement already satisfied: importlib-resources in
/usr/local/lib/python3.10/dist-packages (from prophet) (6.4.0)
Requirement already satisfied: stanio<2.0.0,>=0.4.0 in
/usr/local/lib/python3.10/dist-packages (from cmdstanpy>=1.0.4->prophet) (0.5.0)
Requirement already satisfied: python-dateutil in
/usr/local/lib/python3.10/dist-packages (from holidays>=0.25->prophet) (2.8.2)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=2.0.0->prophet)
(1.2.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-
packages (from matplotlib>=2.0.0->prophet) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=2.0.0->prophet)
(4.53.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=2.0.0->prophet)
(1.4.5)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=2.0.0->prophet) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-
packages (from matplotlib>=2.0.0->prophet) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=2.0.0->prophet)
(3.1.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
```

```
packages (from pandas>=1.0.4->prophet) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-
packages (from pandas>=1.0.4->prophet) (2024.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil->holidays>=0.25->prophet) (1.16.0)
```

The function `sales_pred` predicts future sales for a specific store using the Prophet time series forecasting library. It generates two interactive plots using Plotly:

1. **Sales Forecast:** This plot displays both the historical sales data (as markers) and the forecasted sales (as a line) for the specified store over time. The x-axis represents the date, and the y-axis represents the sales amount. This visualization helps to understand the past sales patterns and the predicted future trends.

2. **Trend and Seasonality Components:** This plot decomposes the sales forecast into its underlying components: the overall trend and the yearly seasonality. The trend component shows the long-term direction of the sales (e.g., increasing or decreasing), while the yearly seasonality component captures any repeating patterns that occur throughout the year (e.g., higher sales during holidays or certain months). Understanding these components provides insights into the factors influencing sales and helps to assess the reliability of the forecast.

How it works:

1. **Filtering:** The function first filters the input sales data (`sales_df`) to isolate the sales information for the specified `Store_Id`.
2. **Preparation:** It then renames the `Date` and `Sales` columns to `ds` and `y`, respectively, to match the format required by Prophet. The data is sorted by date to ensure correct chronological order.
3. **Model Fitting:** A Prophet model is created and fitted to the prepared sales data. This model learns the underlying patterns and trends in the historical sales data.
4. **Forecasting:** The model generates a forecast for the specified number of `periods` into the future.
5. **Visualization:** Plotly is used to create interactive plots that allow the user to zoom, pan, and hover over data points for a detailed view. The first plot displays the historical and forecasted sales, while the second plot visualizes the trend and seasonality components of the forecast.

[38]:

```python
def sales_pred(Store_Id, sales_df, periods):
    from prophet import Prophet
    import plotly.graph_objs as go
    from plotly.subplots import make_subplots
    import plotly.io as pio

    # Filter sales data for the specified Store_Id
    sales_df = sales_df[sales_df['Store'] == Store_Id]
    sales_df = sales_df[['Date', 'Sales']]
    sales_df.rename(columns={'Date': 'ds', 'Sales': 'y'}, inplace=True)
    sales_df = sales_df.sort_values('ds')

    # Initialize Prophet model and fit to data
    m = Prophet()
    m.fit(sales_df)

    # Create future dataframe for forecasting
    future = m.make_future_dataframe(periods=periods)
    forecast = m.predict(future)

    # Plotting with Plotly
    fig = go.Figure()

    # Add the main plot (forecast)
    fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yhat'], mode='lines', name='Forecast'))

    # Add the actual sales data points
    fig.add_trace(go.Scatter(x=sales_df['ds'], y=sales_df['y'], mode='markers', name='Actual Sales'))

    # Update layout with labels and title
    fig.update_layout(title='Sales Forecast',
                      xaxis_title='Date',
                      yaxis_title='Sales')

    # Create components plot using subplot functionality of Plotly
    fig2 = make_subplots(rows=2, cols=1)

    # Add trend and seasonality components to subplot
    fig2.add_trace(go.Scatter(x=forecast['ds'], y=forecast['trend'], mode='lines', name='Trend'), row=1, col=1)
    fig2.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yearly'], mode='lines', name='Yearly Seasonality'), row=2, col=1)

    # Update layout of components plot
    fig2.update_layout(title='Trend and Seasonality Components',
                       xaxis_title='Date',
                       yaxis_title='Component Value')

    # Show the figures directly in the notebook
    pio.show(fig)
    pio.show(fig2)
```

This code calls the previously defined `sales_pred` function to generate a sales forecast for store ID 12. It will predict sales for the next 60 periods (days, weeks, months, etc., depending on the frequency of the data) using the Prophet model. The results will be visualized with two Plotly charts: one showing the forecast along with the actual sales data, and another detailing the trend and seasonality components of the forecast.

[39]:

```
sales_pred(12,data,60)
```

```
INFO:prophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpg0e2dzg1/9s_2wmt9.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpg0e2dzg1/v6f436f3.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-
packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=11936', 'data',
'file=/tmp/tmpg0e2dzg1/9s_2wmt9.json', 'init=/tmp/tmpg0e2dzg1/v6f436f3.json',
'output',
'file=/tmp/tmpg0e2dzg1/prophet_modeliag7fc8z/prophet_model-20240622154039.csv',
'method=optimize', 'algorithm=lbfgs', 'iter=10000']
15:40:39 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
15:40:39 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```

This code defines a function `sales_preds` that predicts future sales for a specific store using the Prophet time series forecasting library, incorporating holidays. It generates two interactive plots using Plotly to visualize the forecast and its components.

Key improvements from the previous version:

- **Holiday Incorporation:** The function now accepts an additional argument `holidays`, which is a DataFrame containing information about holidays that may impact sales. This allows the Prophet model to account for the effects of holidays on sales patterns, potentially leading to more accurate forecasts.
- **No Changes in Plotting:** The visualization part of the code remains the same, providing a clear picture of the actual and predicted sales, as well as the trend and yearly seasonality components.

Overall, this updated function enhances the sales forecasting capabilities by considering the impact of holidays, making it a more comprehensive tool for predicting store sales.

[40]:

```
def sales_preds(Store_Id, sales_df, holidays ,periods):
    from prophet import Prophet
    import plotly.graph_objs as go
    from plotly.subplots import make_subplots
    import plotly.io as pio

    # Filter sales data for the specified Store_Id
    sales_df = sales_df[sales_df['Store'] == Store_Id]
    sales_df = sales_df[['Date', 'Sales']]
    sales_df.rename(columns={'Date': 'ds', 'Sales': 'y'}, inplace=True)
    sales_df = sales_df.sort_values('ds')

    # Initialize Prophet model and fit to data
    m = Prophet(holidays= holidays)
    m.fit(sales_df)

    # Create future dataframe for forecasting
    future = m.make_future_dataframe(periods=periods)
    forecast = m.predict(future)

    # Plotting with Plotly
    fig = go.Figure()

    # Add the main plot (forecast)
    fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yhat'],␣
↪mode='lines', name='Forecast'))

    # Add the actual sales data points
    fig.add_trace(go.Scatter(x=sales_df['ds'], y=sales_df['y'], mode='markers',␣
↪name='Actual Sales'))

    # Update layout with labels and title
    fig.update_layout(title='Sales Forecast',
                      xaxis_title='Date',
                      yaxis_title='Sales')

    # Create components plot using subplot functionality of Plotly
    fig2 = make_subplots(rows=2, cols=1)

    # Add trend and seasonality components to subplot
    fig2.add_trace(go.Scatter(x=forecast['ds'], y=forecast['trend'],␣
↪mode='lines', name='Trend'), row=1, col=1)
    fig2.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yearly'],␣
↪mode='lines', name='Yearly Seasonality'), row=2, col=1)

    # Update layout of components plot
    fig2.update_layout(title='Trend and Seasonality Components',
                       xaxis_title='Date',
                       yaxis_title='Component Value')

    # Show the figures directly in the notebook
    pio.show(fig)
    pio.show(fig2)
```

The code filters the DataFrame `data` to select rows where the `SchoolHoliday` column is equal to 1. Then, it extracts the values from the `Date` column of these filtered rows. The `SchoolHoliday` column indicates whether a particular date is a school holiday (1) or not (0), this code aims to retrieve all the dates in the dataset that are marked as school holidays.

```
[41]: school_holidays = data[data['SchoolHoliday']==1].loc[:, 'Date'].values
      school_holidays
```

```
[41]: array(['2015-07-31', '2015-07-30', '2015-07-29', …, '2013-01-04',
             '2013-01-03', '2013-01-02'], dtype=object)
```

```
[42]: school_holidays.shape
```

```
[42]: (163457,)
```

The code filters the DataFrame `data` to select rows where the `StateHoliday` column is equal to “a”, “b”, or “c”. Then, it extracts the values from the `Date` column of these filtered rows. Assuming that the `StateHoliday` column indicates whether a particular date is a state holiday (“a”, “b”, or “c”) or not (some other value), this code aims to retrieve all the dates in the dataset that are marked as state holidays.

```
[43]: state_holidays = data[(data['StateHoliday']== "a") | (data['StateHoliday']==␣
        ↪"b") | (data['StateHoliday']== "c")].loc[:, 'Date'].values
      state_holidays
```

```
[43]: array(['2014-10-03', '2013-10-03', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2014-06-19', '2013-05-30',
             '2014-10-03', '2013-10-03', '2015-05-01', '2014-10-31',
             '2014-05-01', '2013-10-03', '2013-05-01', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2013-08-15', '2015-06-04', '2014-06-19',
             '2013-05-30', '2013-08-15', '2013-08-15', '2015-05-25',
             '2015-05-14', '2015-05-01', '2015-04-06', '2015-04-03',
             '2015-01-01', '2014-12-26', '2014-12-25', '2014-10-03',
             '2014-06-09', '2014-05-29', '2014-05-01', '2014-04-21',
             '2014-04-18', '2014-01-01', '2013-12-26', '2013-12-25',
             '2013-10-03', '2013-05-20', '2013-05-09', '2013-05-01',
             '2013-04-01', '2013-03-29', '2013-01-01', '2014-10-03',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2013-10-03', '2014-10-03', '2013-10-03',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-05-25',
             '2015-05-14', '2015-04-06', '2014-10-03', '2014-06-09',
             '2014-05-29', '2014-04-21', '2013-10-03', '2013-05-20',
             '2013-05-09', '2013-04-01', '2013-08-15', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-05-25', '2014-06-09',
             '2013-05-20', '2013-08-15', '2013-08-15', '2013-08-15',
             '2015-06-04', '2014-06-19', '2013-05-30', '2013-08-15',
             '2015-05-01', '2014-10-31', '2013-10-31', '2015-05-25',
             '2015-05-14', '2015-04-06', '2014-10-03', '2014-06-09',
             '2014-05-29', '2014-04-21', '2013-10-03', '2013-05-20',
             '2013-05-09', '2013-04-01', '2015-05-01', '2014-05-01',
             '2013-08-15', '2015-06-04', '2015-05-25', '2015-05-14',
             '2015-05-01', '2015-04-06', '2015-04-03', '2015-01-01',
             '2014-12-26', '2014-12-25', '2014-10-03', '2014-06-19',
             '2014-06-09', '2014-05-29', '2014-05-01', '2014-04-21',
             '2014-04-18', '2014-01-01', '2013-12-26', '2013-12-25',
             '2013-10-03', '2013-05-30', '2013-05-20', '2013-05-09',
             '2013-05-01', '2013-04-01', '2013-03-29', '2013-01-01',
             '2015-05-25', '2015-05-14', '2015-05-01', '2015-04-06',
             '2015-04-03', '2015-01-01', '2014-12-26', '2014-12-25',
             '2014-10-03', '2014-06-09', '2014-05-29', '2014-05-01',
             '2014-04-21', '2014-04-18', '2014-01-01', '2013-12-26',
             '2013-12-25', '2013-10-03', '2013-05-20', '2013-05-09',
             '2013-05-01', '2013-04-01', '2013-03-29', '2013-01-01',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2015-05-25',
             '2015-05-14', '2015-05-01', '2015-04-06', '2015-04-03',
             '2015-01-01', '2014-12-26', '2014-12-25', '2014-10-03',
             '2014-06-19', '2014-06-09', '2014-05-29', '2014-05-01',
             '2014-04-21', '2014-04-18', '2014-01-01', '2013-12-26',
             '2013-12-25', '2013-10-03', '2013-05-30', '2013-05-20',
             '2013-05-09', '2013-05-01', '2013-04-01', '2013-03-29',
             '2013-01-01', '2015-06-04', '2014-06-19', '2013-05-30',
             '2013-08-15', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-05-25', '2015-05-14', '2015-04-06', '2014-12-26',
             '2014-10-03', '2014-06-09', '2014-05-29', '2014-04-21',
             '2013-10-03', '2013-05-20', '2013-05-09', '2013-04-01',
             '2014-10-03', '2015-05-25', '2015-05-14', '2015-04-06',
             '2014-10-03', '2014-06-09', '2014-05-29', '2014-04-21',
             '2013-10-03', '2013-05-20', '2013-05-09', '2013-04-01',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2015-05-25',
             '2015-05-14', '2015-05-01', '2015-04-06', '2015-04-03',
             '2015-01-01', '2014-12-26', '2014-12-25', '2014-11-01',
             '2014-10-03', '2014-06-19', '2014-06-09', '2014-05-29',
             '2014-05-01', '2014-04-21', '2014-04-18', '2014-01-01',
             '2013-12-26', '2013-12-25', '2013-11-01', '2013-10-03',
             '2013-05-30', '2013-05-20', '2013-05-09', '2013-05-01',
             '2013-04-01', '2013-03-29', '2013-01-01', '2015-06-04',
             '2015-05-25', '2015-05-14', '2015-05-01', '2015-04-06',
             '2015-04-03', '2015-01-06', '2015-01-01', '2014-12-26',
             '2014-12-25', '2014-11-01', '2014-10-03', '2014-06-19',
             '2014-06-09', '2014-05-29', '2014-05-01', '2014-04-21',
             '2014-04-18', '2014-01-06', '2014-01-01', '2013-12-26',
             '2013-12-25', '2013-11-01', '2013-10-03', '2013-05-30',
             '2013-05-20', '2013-05-09', '2013-05-01', '2013-04-01',
             '2013-03-29', '2013-01-06', '2013-01-01', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2015-05-25', '2015-05-14',
             '2015-05-01', '2015-04-06', '2015-04-03', '2015-01-01',
             '2014-12-26', '2014-12-25', '2014-10-03', '2014-06-09',
             '2014-05-29', '2014-05-01', '2014-04-21', '2014-04-18',
             '2014-01-01', '2013-12-26', '2013-12-25', '2013-10-03',
             '2013-05-20', '2013-05-09', '2013-05-01', '2013-04-01',
             '2013-03-29', '2013-01-01', '2013-08-15', '2015-05-25',
             '2015-05-14', '2015-04-06', '2014-06-09', '2014-04-21',
             '2013-05-20', '2013-04-01', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-05-25', '2015-05-14', '2015-04-06',
             '2014-10-03', '2014-06-09', '2014-05-29', '2014-04-21',
             '2013-10-03', '2013-05-20', '2013-05-09', '2013-04-01',
             '2015-06-04', '2014-06-19', '2013-05-30', '2014-10-31',
             '2013-10-31', '2014-10-31', '2013-10-31', '2013-08-15',
             '2015-05-25', '2015-05-14', '2015-05-01', '2015-04-06',
             '2015-04-03', '2015-01-01', '2014-12-26', '2014-12-25',
             '2014-10-03', '2014-06-09', '2014-05-29', '2014-05-01',
             '2014-04-21', '2014-04-18', '2014-01-01', '2013-12-26',
             '2013-12-25', '2013-10-03', '2013-05-20', '2013-05-09',
             '2013-05-01', '2013-04-01', '2013-03-29', '2013-01-01',
             '2015-06-04', '2015-05-25', '2015-05-14', '2015-05-01',
             '2015-04-06', '2015-04-03', '2015-01-06', '2015-01-01',
             '2014-06-19', '2014-06-09', '2014-05-29', '2014-05-01',
             '2014-04-21', '2014-04-18', '2014-01-06', '2014-01-01',
             '2013-12-26', '2013-12-25', '2013-11-01', '2013-10-03',
             '2013-08-15', '2013-05-30', '2013-05-20', '2013-05-09',
             '2013-05-01', '2013-04-01', '2013-03-29', '2013-01-01',
             '2015-06-04', '2013-08-15', '2013-08-15', '2015-05-25',
             '2015-05-14', '2015-04-06', '2014-10-03', '2014-06-09',
             '2014-05-29', '2014-04-21', '2013-10-03', '2013-05-20',
             '2013-05-09', '2013-04-01', '2014-10-03', '2013-10-03',
             '2015-05-25', '2015-05-14', '2015-04-06', '2015-01-01',
             '2014-12-26', '2014-10-03', '2014-06-09', '2014-05-29',
             '2014-04-21', '2014-01-01', '2013-12-26', '2013-10-03',
             '2013-05-20', '2013-05-09', '2013-04-01', '2013-01-01',
             '2013-08-15', '2015-06-04', '2014-06-19', '2013-05-30',
             '2013-10-03', '2015-05-25', '2015-05-14', '2015-05-01',
             '2015-04-06', '2015-04-03', '2015-01-01', '2014-12-26',
             '2014-12-25', '2014-10-03', '2014-06-09', '2014-05-29',
             '2014-05-01', '2014-04-21', '2014-04-18', '2014-01-01',
             '2013-12-26', '2013-12-25', '2013-10-03', '2013-05-20',
             '2013-05-09', '2013-05-01', '2013-04-01', '2013-03-29',
             '2013-01-01', '2013-08-15', '2015-05-25', '2015-05-14',
             '2015-04-06', '2014-10-03', '2014-06-09', '2014-05-29',
             '2014-04-21', '2013-10-03', '2013-05-20', '2013-05-09',
             '2013-04-01', '2014-06-09', '2013-05-20', '2013-10-03',
             '2013-08-15', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2013-08-15', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2014-06-19', '2013-05-30',
             '2014-05-01', '2013-05-01', '2015-06-04', '2015-05-25',
             '2015-05-14', '2015-05-01', '2015-04-06', '2015-04-03',
             '2015-01-01', '2014-12-26', '2014-12-25', '2014-10-03',
             '2014-06-19', '2014-06-09', '2014-05-29', '2014-05-01',
             '2014-04-21', '2014-04-18', '2014-01-01', '2013-12-26',
             '2013-12-25', '2013-10-03', '2013-05-30', '2013-05-20',
             '2013-05-09', '2013-05-01', '2013-04-01', '2013-03-29',
             '2013-01-01', '2015-05-25', '2015-05-14', '2015-05-01',
             '2015-04-06', '2015-04-03', '2015-01-01', '2014-12-26',
             '2014-12-25', '2014-10-03', '2014-06-09', '2014-05-29',
             '2014-05-01', '2014-04-21', '2014-04-18', '2014-01-01',
             '2013-12-26', '2013-12-25', '2013-10-03', '2013-05-20',
             '2013-05-09', '2013-05-01', '2013-04-01', '2013-03-29',
             '2013-01-01', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2013-08-15',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-05-25',
             '2015-05-14', '2015-04-06', '2014-10-03', '2014-06-09',
             '2014-05-29', '2014-04-21', '2013-10-03', '2013-05-20',
             '2013-05-09', '2013-04-01', '2015-06-04', '2015-05-25',
             '2015-05-14', '2015-05-01', '2015-04-06', '2015-04-03',
             '2015-01-01', '2014-12-26', '2014-12-25', '2014-11-01',
             '2014-10-03', '2014-06-19', '2014-06-09', '2014-05-29',
             '2014-05-01', '2014-04-21', '2014-04-18', '2014-01-01',
             '2013-12-26', '2013-12-25', '2013-11-01', '2013-10-03',
             '2013-05-30', '2013-05-20', '2013-05-09', '2013-05-01',
             '2013-04-01', '2013-03-29', '2013-01-01', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2015-05-25', '2015-05-14',
             '2015-05-01', '2015-04-06', '2015-04-03', '2015-01-01',
             '2014-12-26', '2014-12-25', '2014-11-01', '2014-10-03',
             '2014-06-19', '2014-06-09', '2014-05-29', '2014-05-01',
             '2014-04-21', '2014-04-18', '2014-01-01', '2013-12-26',
             '2013-12-25', '2013-11-01', '2013-10-03', '2013-05-30',
             '2013-05-20', '2013-05-09', '2013-05-01', '2013-04-01',
             '2013-03-29', '2013-01-01', '2014-10-03', '2013-10-03',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-05-25', '2015-05-14', '2015-04-06',
             '2014-10-03', '2014-06-09', '2014-05-29', '2014-04-21',
             '2013-10-03', '2013-05-20', '2013-05-09', '2013-04-01',
             '2015-05-25', '2015-05-14', '2015-04-06', '2014-10-03',
             '2014-06-09', '2014-05-29', '2014-04-21', '2013-10-03',
             '2013-05-20', '2013-05-09', '2013-04-01', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-05-25', '2015-05-14', '2015-04-06',
             '2014-10-03', '2014-06-09', '2014-05-29', '2014-04-21',
             '2013-10-03', '2013-05-20', '2013-05-09', '2013-04-01',
             '2013-08-15', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2015-05-25', '2015-05-14', '2015-05-01', '2015-04-06',
             '2015-04-03', '2015-01-06', '2015-01-01', '2014-12-26',
             '2014-12-25', '2014-11-01', '2014-10-03', '2014-06-19',
             '2014-06-09', '2014-05-29', '2014-05-01', '2014-04-21',
             '2014-04-18', '2014-01-06', '2014-01-01', '2013-12-26',
             '2013-12-25', '2013-11-01', '2013-10-03', '2013-05-30',
             '2013-05-20', '2013-05-09', '2013-05-01', '2013-04-01',
             '2013-03-29', '2013-01-06', '2013-01-01', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2013-08-15', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2014-06-19', '2013-05-30',
             '2015-06-04', '2014-06-19', '2013-05-30', '2015-06-04',
             '2014-06-19', '2013-05-30', '2013-08-15', '2015-06-04',
             '2014-06-19', '2013-05-30', '2013-08-15', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2014-10-03', '2015-05-25', '2015-05-14',
             '2015-04-06', '2014-10-03', '2014-06-09', '2014-05-29',
             '2014-04-21', '2013-10-03', '2013-05-20', '2013-05-09',
             '2013-04-01', '2013-08-15', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-05-25', '2015-05-14', '2015-05-01',
             '2015-04-06', '2015-04-03', '2015-01-01', '2014-12-26',
             '2014-12-25', '2014-10-03', '2014-06-09', '2014-05-29',
             '2014-05-01', '2014-04-21', '2014-04-18', '2014-01-01',
             '2013-12-26', '2013-12-25', '2013-10-03', '2015-06-04',
             '2014-06-19', '2013-05-30', '2014-10-03', '2015-06-04',
             '2014-06-19', '2013-05-30', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-06-04', '2015-05-25', '2015-05-14',
             '2015-05-01', '2015-04-06', '2015-04-03', '2015-01-01',
             '2014-12-26', '2014-12-25', '2014-11-01', '2014-10-03',
             '2014-06-19', '2014-06-09', '2014-05-29', '2014-05-01',
             '2014-04-21', '2014-04-18', '2014-01-01', '2013-12-26',
             '2013-12-25', '2013-11-01', '2013-10-03', '2013-05-30',
             '2013-05-20', '2013-05-09', '2013-05-01', '2013-04-01',
             '2013-03-29', '2013-01-01', '2015-06-04', '2014-06-19',
             '2013-05-30', '2015-05-25', '2015-05-14', '2015-04-06',
             '2014-12-26', '2014-10-03', '2014-06-09', '2014-05-29',
             '2014-04-21', '2013-12-26', '2013-10-03', '2013-05-20',
             '2013-05-09', '2013-04-01', '2013-08-15', '2015-06-04',
             '2014-06-19', '2013-05-30'], dtype=object)
```

```
[44]: state_holidays.shape
```

```
[44]: (910,)
```

This code creates a DataFrame named `state_holidays` with two columns:

1. `ds`: This column contains the dates from the `state_holidays` variable (which we assume is a list or array of date strings) converted to datetime format using `pd.to_datetime`.
2. `holiday`: This column is a constant value of 'state_holiday', indicating that all these dates represent state holidays.

This DataFrame is typically used as input for the Prophet model, allowing it to account for the impact of state holidays on the sales forecasts.

```
[45]: state_holidays = pd.DataFrame({'ds': pd.to_datetime(state_holidays),
                                     'holiday':'state_holiday'})
```

```
[46]: state_holidays
```

```
[46]:             ds        holiday
      0   2014-10-03  state_holiday
      1   2013-10-03  state_holiday
      2   2015-06-04  state_holiday
      3   2014-06-19  state_holiday
      4   2013-05-30  state_holiday
      ..         …              …
      905 2013-04-01  state_holiday
      906 2013-08-15  state_holiday
      907 2015-06-04  state_holiday
      908 2014-06-19  state_holiday
      909 2013-05-30  state_holiday

      [910 rows x 2 columns]
```

This code creates a DataFrame called `school_holidays` with two columns:

1. **ds**: This column contains the dates from the variable `school_holidays`, which is assumed to be a list or array of date strings. These date strings are converted into datetime objects using `pd.to_datetime`.
2. `holiday`: This column has a constant value of 'school_holiday', indicating that all these dates represent school holidays.

This DataFrame is typically used as input for the Prophet model, allowing it to account for the impact of school holidays on the sales forecasts.

[47]:

```
school_holidays = pd.DataFrame({'ds': pd.to_datetime(school_holidays),
                                'holiday':'school_holiday'})
```

[48]:

```
school_holidays
```

[48]:

```
                ds         holiday
0       2015-07-31  school_holiday
1       2015-07-30  school_holiday
2       2015-07-29  school_holiday
3       2015-07-28  school_holiday
4       2015-07-27  school_holiday
…              …             …
163452  2013-02-05  school_holiday
163453  2013-02-04  school_holiday
163454  2013-01-04  school_holiday
163455  2013-01-03  school_holiday
163456  2013-01-02  school_holiday

[163457 rows x 2 columns]
```

This code combines the `state_holidays` and `school_holidays` DataFrames into a single DataFrame called `school_state_holidays`. Since both DataFrames share the same column structure (they both have `ds` and `holiday` columns), this effectively appends the rows of `school_holidays` to the end of `state_holidays`, creating a combined list of all state and school holidays. This combined DataFrame can then be used as input for the Prophet model to account for the impact of both types of holidays on sales forecasts.

[49]:

```
school_state_holidays = pd.concat([state_holidays,school_holidays])
```

[50]:

```
school_state_holidays
```

[50]:

```
                ds         holiday
0       2014-10-03   state_holiday
1       2013-10-03   state_holiday
2       2015-06-04   state_holiday
3       2014-06-19   state_holiday
4       2013-05-30   state_holiday
…              …             …
163452  2013-02-05  school_holiday
```

```
 163453 2013-02-04  school_holiday
 163454 2013-01-04  school_holiday
 163455 2013-01-03  school_holiday
 163456 2013-01-02  school_holiday

 [164367 rows x 2 columns]
```

This code calls the function `sales_preds` to predict sales for store ID 12 over the next 90 periods, taking into account both school and state holidays. Here's what it does:

1. **Store ID (12):** Specifies the store for which the prediction is made.
2. **Data (`data`):** Provides the historical sales data required for training the Prophet model.
3. **Holidays (`school_state_holidays`):** This DataFrame includes dates and types of holidays (school or state) that can potentially impact sales.
4. **Periods (90):** Sets the number of periods (days, weeks, months, etc.) into the future to forecast.

The function will then:

1. Filter the `data` for store ID 12.
2. Preprocess the data for Prophet (rename columns, sort by date).
3. Initialize a Prophet model and include the `school_state_holidays` data to account for holidays.
4. Fit the model to the data.
5. Make predictions for the next 90 periods.
6. Generate two interactive Plotly plots: one showing the forecast and actual sales, and another displaying the trend and seasonality components of the forecast.

[55]:

```
sales_preds(12,data,school_state_holidays,1095)
```

```
INFO:prophet:Disabling daily seasonality. Run prophet with
daily_seasonality=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpg0e2dzg1/g1aqcobo.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpg0e2dzg1/3kfrph8k.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-
packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=43005', 'data',
'file=/tmp/tmpg0e2dzg1/g1aqcobo.json', 'init=/tmp/tmpg0e2dzg1/3kfrph8k.json',
'output',
'file=/tmp/tmpg0e2dzg1/prophet_modelnvylm_mt/prophet_model-20240622173049.csv',
'method=optimize', 'algorithm=lbfgs', 'iter=10000']
17:30:49 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
17:30:49 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```