CAPSTONE= KLYDE CARPIZO

A COMPREHENSIVE SALES ANALYTICS DASHBOARD CODE:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from ipywidgets import widgets
from IPython.display import display, clear_output

# Create sample sales data
# In a real scenario, you would load data from a CSV or database
data = {
    'Date': pd.date_range(start='2023-01-01', periods=365, freq='D'),
    'Region': ['North', 'South', 'East', 'West', 'Central'] * 73,
    'Product_Category': ['Electronics', 'Clothing', 'Furniture', 'Groceries', 'Beauty'] * 73,
    'Sales_Amount': [round(100 + abs(i % 15) * 50 + abs((i % 7) * 30), 2) for i in range(365)],
    'Profit_Margin': [round(0.1 + abs(i % 10) * 0.02, 2) for i in range(365)],
    'Customer_Satisfaction': [round(3 + abs(i % 5) * 0.5, 1) for i in range(365)],
    'Sales_Representative': ['John Doe', 'Jane Smith', 'Bob Johnson', 'Alice Brown', 'Charlie
Wilson'] * 73
}

df = pd.DataFrame(data)

# Extract month and year for temporal analysis
df['Month'] = df['Date'].dt.month_name()
df['Year'] = df['Date'].dt.year
df['Month_Num'] = df['Date'].dt.month  # For sorting

# Create dashboard components
def create_sales_comparison(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Group by region for comparison
    region_sales = filtered_df.groupby('Region')['Sales_Amount'].sum().reset_index()

    fig = px.bar(
        region_sales,
        x='Region',
        y='Sales_Amount',
```

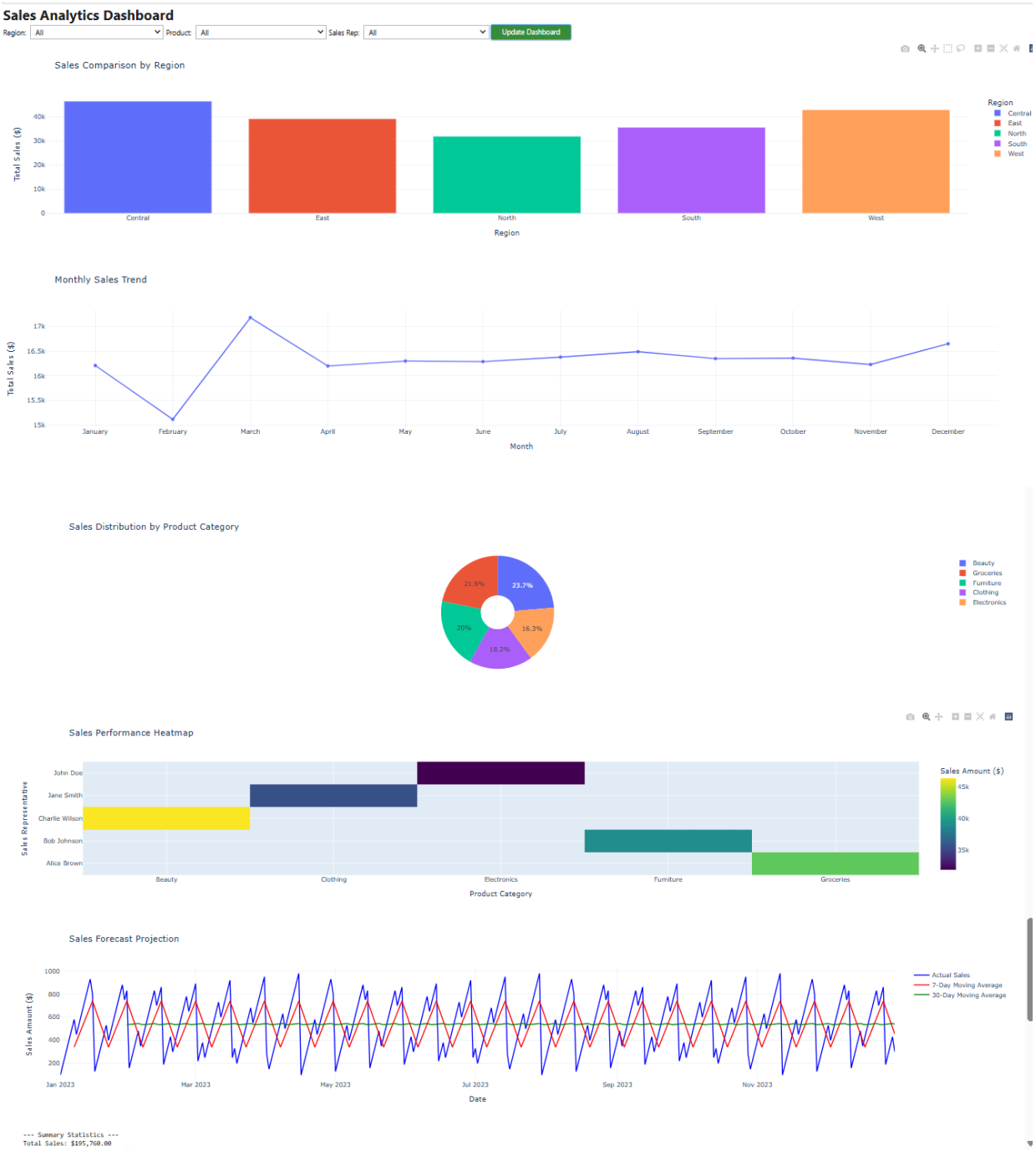
```
        title='Sales Comparison by Region',
        labels={'Sales_Amount': 'Total Sales ($)', 'Region': 'Region'},
        color='Region',
        template='plotly_white'
    )

    fig.update_layout(height=400)
    return fig

def create_sales_trend(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Group by month for trend analysis
    monthly_sales = filtered_df.groupby(['Year', 'Month_Num',
'Month'])['Sales_Amount'].sum().reset_index()
    monthly_sales = monthly_sales.sort_values(['Year', 'Month_Num'])

    fig = px.line(
        monthly_sales,
        x='Month',
        y='Sales_Amount',
        title='Monthly Sales Trend',
        labels={'Sales_Amount': 'Total Sales ($)', 'Month': 'Month'},
        markers=True,
        template='plotly_white'
    )

    fig.update_layout(height=400, xaxis_title='Month')
    return fig

def create_regional_distribution(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Group by product category
    product_sales = filtered_df.groupby('Product_Category')['Sales_Amount'].sum().reset_index()

    fig = px.pie(
        product_sales,
        values='Sales_Amount',
        names='Product_Category',
        title='Sales Distribution by Product Category',
        template='plotly_white',
        hole=0.3
    )
```

```
    fig.update_layout(height=400)
    return fig

def create_performance_heatmap(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Create a pivot table for the heatmap
    pivot_data = filtered_df.pivot_table(
        index='Sales_Representative',
        columns='Product_Category',
        values='Sales_Amount',
        aggfunc='sum'
    )

    fig = go.Figure(data=go.Heatmap(
        z=pivot_data.values,
        x=pivot_data.columns,
        y=pivot_data.index,
        colorscale='Viridis',
        colorbar=dict(title='Sales Amount ($)')
    ))

    fig.update_layout(
        title='Sales Performance Heatmap',
        xaxis_title='Product Category',
        yaxis_title='Sales Representative',
        height=400
    )

    return fig

def create_forecast_projection(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Group by date for time series analysis
    daily_sales = filtered_df.groupby('Date')['Sales_Amount'].sum().reset_index()

    # Simple moving average forecast (in a real scenario, you'd use more advanced models)
    daily_sales['MA7'] = daily_sales['Sales_Amount'].rolling(window=7).mean()
    daily_sales['MA30'] = daily_sales['Sales_Amount'].rolling(window=30).mean()

    fig = go.Figure()
```

```
    fig.add_trace(go.Scatter(
        x=daily_sales['Date'],
        y=daily_sales['Sales_Amount'],
        mode='lines',
        name='Actual Sales',
        line=dict(color='blue')
    ))

    fig.add_trace(go.Scatter(
        x=daily_sales['Date'],
        y=daily_sales['MA7'],
        mode='lines',
        name='7-Day Moving Average',
        line=dict(color='red')
    ))

    fig.add_trace(go.Scatter(
        x=daily_sales['Date'],
        y=daily_sales['MA30'],
        mode='lines',
        name='30-Day Moving Average',
        line=dict(color='green')
    ))

    fig.update_layout(
        title='Sales Forecast Projection',
        xaxis_title='Date',
        yaxis_title='Sales Amount ($)',
        height=400,
        template='plotly_white'
    )

    return fig

def filter_dataframe(df, region=None, product=None, sales_rep=None):
    filtered_df = df.copy()

    if region and region != 'All':
        filtered_df = filtered_df[filtered_df['Region'] == region]

    if product and product != 'All':
        filtered_df = filtered_df[filtered_df['Product_Category'] == product]

    if sales_rep and sales_rep != 'All':
```

```
        filtered_df = filtered_df[filtered_df['Sales_Representative'] == sales_rep]

    return filtered_df

# Create interactive widgets for filtering
region_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Region'].unique()),
    value='All',
    description='Region:',
    style={'description_width': 'initial'}
)

product_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Product_Category'].unique()),
    value='All',
    description='Product:',
    style={'description_width': 'initial'}
)

sales_rep_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Sales_Representative'].unique()),
    value='All',
    description='Sales Rep:',
    style={'description_width': 'initial'}
)

update_button = widgets.Button(
    description='Update Dashboard',
    button_style='success',
    tooltip='Click to update the dashboard with selected filters'
)

output = widgets.Output()

# Define update function
def update_dashboard(b):
    with output:
        clear_output()

        # Get current filter values
        region = region_dropdown.value
        product = product_dropdown.value
        sales_rep = sales_rep_dropdown.value
```

```
        # Create figures
        fig1 = create_sales_comparison(df, region, product, sales_rep)
        fig2 = create_sales_trend(df, region, product, sales_rep)
        fig3 = create_regional_distribution(df, region, product, sales_rep)
        fig4 = create_performance_heatmap(df, region, product, sales_rep)
        fig5 = create_forecast_projection(df, region, product, sales_rep)

        # Display the figures
        fig1.show()
        fig2.show()
        fig3.show()
        fig4.show()
        fig5.show()

        # Display summary statistics
        filtered_df = filter_dataframe(df, region, product, sales_rep)

        print("\n--- Summary Statistics ---")
        print(f"Total Sales: ${filtered_df['Sales_Amount'].sum():,.2f}")
        print(f"Average Profit Margin: {filtered_df['Profit_Margin'].mean():.2%}")
        print(f"Average Customer Satisfaction:
{filtered_df['Customer_Satisfaction'].mean():.1f}/5.0")

# Connect the button to the update function
update_button.on_click(update_dashboard)

# Create the dashboard layout
dashboard_title = widgets.HTML("<h1>Sales Analytics Dashboard</h1>")
filter_widgets = widgets.HBox([region_dropdown, product_dropdown, sales_rep_dropdown,
update_button])
dashboard = widgets.VBox([dashboard_title, filter_widgets, output])

# Display the dashboard
display(dashboard)

# Initialize the dashboard with default values
update_dashboard(None)
```




```
--- Summary Statistics ---
Total Sales: $195,760.00
Average Profit Margin: 18.93%
Average Customer Satisfaction: 4.0/5.0
```

KEY TAKEAWAYS AND EXPLANATION:

1. **LIBRARIES**
   Import the Libraries we need for the project:
   - pandas - the most basic python library to use for data manipulation, this library provides a DataFrame object, creating tables and functions that can analyze and handle data.We will need it for reading, cleaning, and transforming our sales data.
   - matplotlib.pyplot - a primary library for plots
   - seaborn - Built on top of matplotlib on which provides a high-level interface creating statistical and visual plots
   - plotly.express - greate for creating charts
   - plotly.graph objects - provides more control over structure and appearance giving plotly charts better visuals.
   - ipywidgets - an essential creative tool that creates dropdown menus, and visual buttons. These widgets help users to interact with the dash board.
   - IPython.display - This tool is use for displaying content of the data and clearing output area

2. **THE DATA**

```
# Create sample sales data
# In a real scenario, you would load data from a CSV or database
data = {
    'Date': pd.date_range(start='2023-01-01', periods=365, freq='D'),
    'Region': ['North', 'South', 'East', 'West', 'Central'] * 73,
    'Product_Category': ['Electronics', 'Clothing', 'Furniture', 'Groceries', 'Beauty'] * 73,
    'Sales_Amount': [round(100 + abs(i % 15) * 50 + abs((i % 7) * 30), 2) for i in range(365)],
    'Profit_Margin': [round(0.1 + abs(i % 10) * 0.02, 2) for i in range(365)],
    'Customer_Satisfaction': [round(3 + abs(i % 5) * 0.5, 1) for i in range(365)],
    'Sales_Representative': ['John Doe', 'Jane Smith', 'Bob Johnson', 'Alice Brown', 'Charlie Wilson'] * 73
}
```

As it is noted, usually we upload the data from a CSV file or database. But let's create data out of scratch and patch it in our database directly.

Let's make at least 3 key elements in our data:

- The keys are column names are:
  *Date - uses pd.date_range() to generate a sequence of 365 days, starting fro '2023-01-01', with a daily frequency code ('D')
  *Region, Product_Category, Sales_Representative - these lists are repeated to create enough entries for each day of the year. For example, `['North', 'South', 'East', 'West', 'Central'] \* 73` repeats the list 73 times to get 365 entries (5 regions x 73 = 365 days).

*Sales_Amount, Profit_Margin, Customer_Satisfaction - these list generated list comprehensions with concise list targets.The formulas inside the list comprehensions create some variability in the data. Let's break down `'Sales_Amount'`:

- *`[... for i in range(365)]`: This loops 365 times (once for each day).
- *`100 + abs(i % 15) \* 50 + abs((i % 7) \* 30)`: This is the formula to calculate the sales amount for each day:
- `i % 15`: The modulo operator (`%`) gives the remainder when `i` is divided by 15. This creates a cyclical pattern with a period of 15.
- `abs(...)`: The absolute value ensures the result is positive.
- The formula combines two cyclical patterns (one with a period of 15, one with a period of 7) and adds them to a base value of 100. This creates some realistic-looking variation in the sales data.
- `round(..., 2)`: This rounds the sales amount to 2 decimal places.

3. **DATA PROCESSING:**

```
df = pd.DataFrame(data)

# Extract month and year for temporal analysis
df['Month'] = df['Date'].dt.month_name()
df['Year'] = df['Date'].dt.year
df['Month_Num'] = df['Date'].dt.month  # For sorting
```

df = pd. DataFrame(data) line creates a pandas DataFrame shortcut named df from the data dictionary. The DataFrame is the data structure that pandas uses to organize our data through rows and columns

Next, we extract our data through sections in 3 new columns:

- df['Month'] = df['Date'].dt.month_name(): Extracts the month name from the 'Date' column and create a new column 'Month'.Then `.dt` is used to access datetime properties of the 'Date' column.
- `df['Year'] = df['Date'].dt.year`: Extracts the year from the 'Date' column and creates a new column called 'Year'.
- `df['Month_Num'] = df['Date'].dt.month`: Extracts the numerical month (1 for January, 2 for February, etc.) from the 'Date' column and creates a new

column called 'Month_Num'. This is useful for sorting the months correctly (e.g., when creating the 'Monthly Sales Trend' chart).

4. **DEFINING DASHBOARD COMPONENTS(OUR FUNCTIONS)**
#Note: This is the hardest part and longest to execute the code

This is the core of the dashboard, where each visualization is created.

- **`create_sales_comparison(df, region=None, product=None, sales_rep=None)`**:
    - Filters the DataFrame using the `filter_dataframe` function (explained later).
    - Groups the filtered data by 'Region' and calculates the sum of 'Sales_Amount' for each region.
    - Creates a bar chart using `plotly.express` (`px.bar`) to compare sales across regions.
    - Customizes the chart with a title, labels, and color.
    - Returns the `fig` object, which can then be displayed.
    - ```python
      def create_sales_comparison(df, region=None, product=None, sales_rep=None):
          filtered_df = filter_dataframe(df, region, product, sales_rep)

          # Group by region for comparison
          region_sales = filtered_df.groupby('Region')['Sales_Amount'].sum().reset_index()

          fig = px.bar(
              region_sales,
              x='Region',
              y='Sales_Amount',
              title='Sales Comparison by Region',
              labels={'Sales_Amount': 'Total Sales ($)', 'Region': 'Region'},
              color='Region',
              template='plotly_white'
          )

          fig.update_layout(height=400)
          return fig
      ```

- **`create_sales_trend(df, region=None, product=None, sales_rep=None)`**:
    - Filters the DataFrame.
    - Groups the data by 'Year', 'Month_Num', and 'Month' and calculates the sum of 'Sales_Amount'.
    - Sorts the data by 'Year' and 'Month_Num' to ensure the months are in the correct order.
    - Creates a line chart using `plotly.express` (`px.line`) to show the trend of sales over time.
    - Customizes the chart with a title, labels, and markers.
    - Returns the `fig` object.
      ```

```
def create_sales_trend(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Group by month for trend analysis
    monthly_sales = filtered_df.groupby(['Year', 'Month_Num', 'Month'])['Sales_Amount'].sum().reset_index()
    monthly_sales = monthly_sales.sort_values(['Year', 'Month_Num'])

    fig = px.line(
        monthly_sales,
        x='Month',
        y='Sales_Amount',
        title='Monthly Sales Trend',
        labels={'Sales_Amount': 'Total Sales ($)', 'Month': 'Month'},
        markers=True,
        template='plotly_white'
    )

    fig.update_layout(height=400, xaxis_title='Month')
    return fig
```

- **`create_regional_distribution(df, region=None, product=None, sales_rep=None)`**:
  - Filters the DataFrame.
  - Groups the data by 'Product_Category' and calculates the sum of 'Sales_Amount'.
  - Creates a pie chart using `plotly.express` (`px.pie`) to show the distribution of sales across product categories.
  - Customizes the chart with a title, values, and labels.
  - Uses `hole` parameter to create a donut chart.
  - Returns the `fig` object.

```
def create_regional_distribution(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Group by product category
    product_sales = filtered_df.groupby('Product_Category')['Sales_Amount'].sum().reset_index()

    fig = px.pie(
        product_sales,
        values='Sales_Amount',
        names='Product_Category',
        title='Sales Distribution by Product Category',
        template='plotly_white',
        hole=0.3
    )

    fig.update_layout(height=400)
    return fig
```

- **`create_performance_heatmap(df, region=None, product=None, sales_rep=None)`**:
    - Filters the DataFrame.
    - Creates a pivot table using `df.pivot_table()`. A pivot table is a way to reshape the data, making it suitable for a heatmap. In this case, it shows the sum of 'Sales_Amount' for each 'Sales_Representative' and 'Product_Category' combination.
    - Creates a heatmap using `plotly.graph_objects` (`go.Figure` and `go.Heatmap`). `plotly.graph_objects` is used here because it provides more control over the heatmap's appearance.
    - Customizes the chart with a title, labels, and color scale.
    - Returns the `fig` object.

```
def create_performance_heatmap(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Create a pivot table for the heatmap
    pivot_data = filtered_df.pivot_table(
        index='Sales_Representative',
        columns='Product_Category',
        values='Sales_Amount',
        aggfunc='sum'
    )

    fig = go.Figure(data=go.Heatmap(
        z=pivot_data.values,
        x=pivot_data.columns,
        y=pivot_data.index,
        colorscale='Viridis',
        colorbar=dict(title='Sales Amount ($)')
    ))

    fig.update_layout(
        title='Sales Performance Heatmap',
        xaxis_title='Product Category',
        yaxis_title='Sales Representative',
        height=400
    )

    return fig
```

- **`create_forecast_projection(df, region=None, product=None, sales_rep=None)`**:
    - Filters the DataFrame.
    - Groups the data by 'Date' and calculates the sum of 'Sales_Amount'.
    - Calculates the 7-day and 30-day moving averages using the `.rolling()` method. Moving averages smooth out short-term fluctuations and can help to see trends.
    - Creates a line chart using `plotly.graph_objects` to display the actual sales and the moving averages.

- Customizes the chart with a title, labels, and line colors.
- Returns the `fig` object.

```
def create_forecast_projection(df, region=None, product=None, sales_rep=None):
    filtered_df = filter_dataframe(df, region, product, sales_rep)

    # Group by date for time series analysis
    daily_sales = filtered_df.groupby('Date')['Sales_Amount'].sum().reset_index()

    # Simple moving average forecast (in a real scenario, you'd use more advanced models)
    daily_sales['MA7'] = daily_sales['Sales_Amount'].rolling(window=7).mean()
    daily_sales['MA30'] = daily_sales['Sales_Amount'].rolling(window=30).mean()

    fig = go.Figure()

    fig.add_trace(go.Scatter(
        x=daily_sales['Date'],
        y=daily_sales['Sales_Amount'],
        mode='lines',
        name='Actual Sales',
        line=dict(color='blue')
    ))

    fig.add_trace(go.Scatter(
        x=daily_sales['Date'],
        y=daily_sales['MA7'],
        mode='lines',
        name='7-Day Moving Average',
        line=dict(color='red')
    ))

    fig.add_trace(go.Scatter(
        x=daily_sales['Date'],
        y=daily_sales['MA30'],
        mode='lines',
        name='30-Day Moving Average',
        line=dict(color='green')
    ))

    fig.update_layout(
        title='Sales Forecast Projection',
        xaxis_title='Date',
        yaxis_title='Sales Amount ($)',
        height=400,
        template='plotly_white'
    )

    return fig
```

- **`filter_dataframe(df, region=None, product=None, sales_rep=None)`**:
  - This is a helper function used by the other functions to filter the DataFrame based on user selections.
  - It takes the DataFrame `df` and optional filter values for `region`, `product`, and `sales_rep`.
  - It creates a copy of the DataFrame to avoid modifying the original.

- It applies the filters if they are not 'All'.
- It returns the filtered DataFrame.

```
def filter_dataframe(df, region=None, product=None, sales_rep=None):
    filtered_df = df.copy()

    if region and region != 'All':
        filtered_df = filtered_df[filtered_df['Region'] == region]

    if product and product != 'All':
        filtered_df = filtered_df[filtered_df['Product_Category'] == product]

    if sales_rep and sales_rep != 'All':
        filtered_df = filtered_df[filtered_df['Sales_Representative'] == sales_rep]

    return filtered_df

# Create interactive widgets for filtering
region_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Region'].unique()),
    value='All',
    description='Region:',
    style={'description_width': 'initial'}
)

product_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Product_Category'].unique()),
    value='All',
    description='Product:',
    style={'description_width': 'initial'}
)

sales_rep_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Sales_Representative'].unique()),
    value='All',
    description='Sales Rep:',
    style={'description_width': 'initial'}
)

update_button = widgets.Button(
    description='Update Dashboard',
    button_style='success',
    tooltip='Click to update the dashboard with selected filters'
)

output = widgets.Output()
```

-

5.CREATING THE WIDGETS:

```
# Create interactive widgets for filtering
region_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Region'].unique()),
    value='All',
    description='Region:',
    style={'description_width': 'initial'}
)

product_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Product_Category'].unique()),
    value='All',
    description='Product:',
    style={'description_width': 'initial'}
)

sales_rep_dropdown = widgets.Dropdown(
    options=['All'] + list(df['Sales_Representative'].unique()),
    value='All',
    description='Sales Rep:',
    style={'description_width': 'initial'}
)

update_button = widgets.Button(
    description='Update Dashboard',
    button_style='success',
    tooltip='Click to update the dashboard with selected filters'
)

output = widgets.Output()
```

This part creates the interactive controls that the user will use to filter the data:

- **`region_dropdown`, `product_dropdown`, `sales_rep_dropdown`**: These create dropdown menus using `widgets.Dropdown`.
    - `options`: Sets the options in the dropdown. It starts with 'All' and then adds the unique values from the corresponding column in the DataFrame (e.g., all the unique regions from `df['Region']`).
    - `value`: Sets the default selected value ('All').
    - `description`: Sets the label for the dropdown.
    - `style`: Adjusts the width of the description label.
- **`update_button`**: Creates a button using `widgets.Button`.
    - `description`: Sets the text on the button.
    - `button_style`: Sets the button style (e.g., 'success' for a green button).
    - `tooltip`: Sets the text that appears when you hover over the button.
- **`output`**: Creates an output area using `widgets.Output()`. This is a region in the Jupyter Notebook where the plots and summary statistics will be displayed.

6. THE UPDATE DASHBOARD FUNCTION

```
# Define update function
def update_dashboard(b):
    with output:
        clear_output()

        # Get current filter values
        region = region_dropdown.value
        product = product_dropdown.value
        sales_rep = sales_rep_dropdown.value

        # Create figures
        fig1 = create_sales_comparison(df, region, product, sales_rep)
        fig2 = create_sales_trend(df, region, product, sales_rep)
        fig3 = create_regional_distribution(df, region, product, sales_rep)
        fig4 = create_performance_heatmap(df, region, product, sales_rep)
        fig5 = create_forecast_projection(df, region, product, sales_rep)

        # Display the figures
        fig1.show()
        fig2.show()
        fig3.show()
        fig4.show()
        fig5.show()

        # Display summary statistics
        filtered_df = filter_dataframe(df, region, product, sales_rep)

        print("\n--- Summary Statistics ---")
        print(f"Total Sales: ${filtered_df['Sales_Amount'].sum():,.2f}")
        print(f"Average Profit Margin: {filtered_df['Profit_Margin'].mean():.2%}")
        print(f"Average Customer Satisfaction: {filtered_df['Customer_Satisfaction'].mean():.1f}/5.0")
```

This function is the heart of the interactivity. It's called when the user clicks the "Update Dashboard" button:

- **`def update_dashboard(b)`**: The `b` argument represents the button that was clicked (it's a convention in ipywidgets).
- **`with output:`**: This is a context manager. Anything that happens inside the `with output:` block will be displayed in the output area (`output`) that we created earlier.
- **`clear_output()`**: This is crucial. It clears any previous content in the output area before displaying the new plots. This prevents the plots from accumulating every time the button is clicked.
- **Get Filter Values**: It gets the current values of the dropdown menus using `.value` (e.g., `region_dropdown.value`).
- **Create Figures**: It calls the functions we defined earlier (`create_sales_comparison`, `create_sales_trend`, etc.) to generate the plots, passing in the DataFrame and the filter values.

- **Display Figures**: It uses `fig.show()` to display each of the generated Plotly figures in the output area.
- **Display Summary Statistics**:
    - It calls `filter_dataframe` to get the filtered data.
    - It calculates and prints some summary statistics using pandas functions:
        - `filtered_df['Sales_Amount'].sum()`: Calculates the total sales amount.
        - `filtered_df['Profit_Margin'].mean()`: Calculates the average profit margin.
        - `filtered_df['Customer_Satisfaction'].mean()`: Calculates the average customer satisfaction.
    - The `f-strings` are used to format the output and include the calculated values in the printed text.

7.CONNECTING WIDGETS AND FUNCTION

```
# Connect the button to the update function
update_button.on_click(update_dashboard)
```

- This line connects the `update_button` to the `update_dashboard` function.
- `.on_click(update_dashboard)`: This tells ipywidgets to call the `update_dashboard` function when the button is clicked. The `update_dashboard` function will then update the plots and summary statistics based on the selected filter values.

8.CREATING THE DASHBOARD LAYOUT

```
# Create the dashboard layout
dashboard_title = widgets.HTML("<h1>Sales Analytics Dashboard</h1>")
filter_widgets = widgets.HBox([region_dropdown, product_dropdown, sales_rep_dropdown, update_button])
dashboard = widgets.VBox([dashboard_title, filter_widgets, output])
```

This part defines how the dashboard is arranged:

- `dashboard_title`: Creates an HTML widget to display the title of the dashboard.
- `filter_widgets`: Creates a horizontal box (`widgets.HBox`) to arrange the dropdown menus and the update button in a row.
- `dashboard`: Creates a vertical box (`widgets.VBox`) to arrange the title, the filter widgets, and the output area in a column. This is the main container for the dashboard.

9.DISPLAY THE DASHBOARD

```
# Display the dashboard
display(dashboard)

# Initialize the dashboard with default values
update_dashboard(None)
```

- `display(dashboard)`: This line displays the entire dashboard (the `widgets.VBox` containing all the elements) in the Jupyter Notebook.
- `update_dashboard(None)`: This line calls the `update_dashboard` function when the code is first executed to display the initial dashboard with all the data and the default filter values. The `None` argument is passed, and the function is written to handle this case.