Building an Interactive EDA Dashboard

for Customer Churn Dataset

by Jenny Zhu MADS Student @ UMICH

Visualization Technique

Narrative Description of Each Visualization Type

1. **Bar Chart**:
   - A bar chart is used to compare categorical data. Each bar's length represents the value or frequency of the corresponding category.
   - This visualization is helpful when analyzing across different categories (e.g., the frequency of observations per group).
2. **Histogram**:
   - A histogram displays the distribution of numeric data as continuous intervals (bins). The height of each bar represents the count or density of data points within the interval.
   - A histogram allows you to assess the spread, central tendency, skewness, and outliers in numerical data.
3. **Boxplot**:
   - A boxplot summarizes the distribution of data by showing the median, quartiles, and potential outliers in one compact chart. Each "box" represents the interquartile range (IQR).
   - This visualization works well for identifying how data values vary among different groups and for spotting outliers.
4. **Scatterplot**:
   - A scatterplot shows the relationship between two continuous variables by marking data points on an x-y axis.
   - It is useful for identifying correlations, trends, clusters, or outliers between variables.

How These Visualizations Complement Each Other

- Together, these visualizations provide a comprehensive understanding of the data by analyzing different aspects:
  - **Bar Chart**: Works with categorical data to provide summaries and group-level insights.
  - **Histogram**: Provides insight into numerical data distributions without focusing on specific relationships.
  - **Boxplot**: Builds on insights from both histograms and bar charts by summarizing data distributions while emphasizing variability and outliers. It can also emphasize side-by-side comparison among different categories.
  - **Scatterplot**: Complements the other charts by showing relationships between two variables, enabling a deeper trend or pattern analysis.

When Each Visualization Should Be Used

- **Bar Chart**: Use it when comparing frequencies or values across categories, e.g., sales per product category.
- **Histogram**: Use it for understanding the distribution of continuous data, e.g., customer spending habits.
- **Boxplot**: Use it when comparing variability, especially across groups, or spotting outliers, e.g., income levels in different customer segments.
- **Scatterplot**: Use it to see the relationship between two variables, e.g., the correlation between marketing budget and sales revenue.

Dashboard-specific considerations

Interactivity Options:

- Enable **filtering and selection**: Allow users to interact with widgets or dropdown filters to select specific categories, variables, or ranges of interest.
- Use **tooltips**: Show additional information (e.g., exact values) when a user hovers over specific points on a scatterplot or bars in a bar chart.
- Add **zoom and pan functionality**: Particularly useful for scatterplots and histograms to explore data points in more detail.

Visualization Library

Framework and Libraries Used

1. **Panel**:
   - **What it is**: Panel is an open-source dashboarding library provided by the `HoloViz` ecosystem, designed for creating interactive and highly customizable dashboards. It supports rendering visualizations, widgets, and layouts in Jupyter Notebooks or standalone web applications.
   - **Who created it**: It is developed and maintained by the HoloViz community.
   - **Why it's suitable**: Panel provides an intuitive way to create interactive applications, supports a wide range of plotting libraries, and integrates seamlessly with Jupyter Notebooks. Additionally, its support for widgets makes it a great choice for enabling interactivity.
2. **HvPlot**:
   - **What it is**: HvPlot is a high-level plotting library that integrates Pandas-like APIs for creating easy, declarative visualizations.
   - **Why it's suitable**: HvPlot provides simplified syntax for exploratory data analysis, enabling you to generate bokeh-backed plots directly from a Pandas DataFrame or other data analysis libraries like Dask or Xarray. Its integration with HoloViews improves interactivity and advanced plotting features.
3. **Other Libraries**:
   - **Pandas**: Used for data manipulation and analysis.
   - **NumPy**: For numerical operations and advanced data computations.
   - **Scipy**: Provides powerful statistical analysis functions (e.g., `stats` module for analysis like distributions).

Benefits and Limitations of the Framework

Benefits:

1. **Easy to Use Approach**: Panel and HvPlot is easy to use. For instance, HvPlot allows you to generate visualizations with a single statement (e.g., `df.hvplot()`).
2. **Jupyter Notebook Integration**: Both Panel and HvPlot integrate seamlessly into Jupyter Notebooks. The `pn.extension()` enables displaying Panel objects, widgets, and interactive features directly in notebooks for exploratory or static analysis.
3. **Support for Interactivity**: Panel enables creating rich, interactive widgets such as dropdowns, sliders, and buttons, making it highly suitable for creating dashboards with dynamic data exploration and user inputs experience.
4. **Display Dashboard on a Local Host Webpage**:

   My favorite feature of this framework is that the final dashboard is displayed on a webpage, although it's locally hosted, it's very neat to get everything in one place and easy to share with audience.

Limitations of the Framework:

1. **Complexity for Beginners**: While HvPlot simplifies basic plotting tasks, Panel's layout system and widget bindings can be overwhelming for users new to declarative-style frameworks. Use needs to equip with essential Python programming skills such as creating classes and functions. It will be even more difficult if user wants to create a reusable code snippets.
2. **Performance for Large Data**:
   - Interactivity and responsiveness can degrade when handling very large datasets, that's why I have to sample certain amount of data in first step.
3. **Tooling and Debugging**:
   - Debugging complex interactions between widgets and plots can sometimes be challenging, as all elements are linked to each other tightly. One slight oversight in one variable would cause big impact on final results.

Why I Chose This Framework

Panel is a great tool for building interactive dashboards, especially for small-scale data visualizations. It's easy to use in Jupyter, allowing you to quickly prototype and tweak interactive features without needing a complex web setup. It integrates seamlessly with popular Python libraries like Pandas, HvPlot, NumPy, and SciPy, making it a powerful choice for exploratory data analysis (EDA) and creating rich, interactive plots. As an open-source tool with strong

community support, I can get any help with debugging if needed. It also offers a variety of interactive features, such as hover effects, tooltips, legends, and widgets like customized sliders and dropdowns, making it ideal for creating dynamic and engaging dashboards for viewers with any levels of data science knowledge.

Install essential libraries

```
!pip install panel hvplot pandas numpy
```

In [1]:
```
# Next, import them into this notebook
import panel as pn
import hvplot.pandas
import pandas as pd
import numpy as np
import holoviews as hv
from scipy import stats

# enable Jupyter to display Panel objects and interactive widgets
pn.extension()
```

In [2]:
```
import logging
import param

# Suppress warnings by setting param's logging level to ERROR
logging.getLogger('param').setLevel(logging.ERROR)

import warnings
warnings.simplefilter(action='ignore', category=UserWarning)
```

Demonstration

In this section, I will demonstrate how to build a compelling interactive dashboard to display information and insights for `customer_churn.csv` dataset step by step:

1. Load, clean and explore the dataset with Pandas.
2. Create individual plots with interactivity
3. Combine plots from step 2 and build a tabbed dashboard

The dataset we are going to use is downloaded from Kaggle by clicking here. I chose this dataset because, in a customer-facing role, understanding **customer churn** is crucial. Customer churn refers to the phenomenon where customers discontinue their relationship or subscription with a company or service provider. This is an extremely important metric for companies looking to stay profitable since churn directly impacts **revenue**, **growth**, and **customer retention.** This dataset presents an excellent opportunity to explore patterns and derive valuable insights. By analyzing and visualizing various variables such as:

- **Numerical Variables**: `['Age', 'Tenure', 'Usage Frequency', 'Support Calls', 'Payment Delay', 'Total Spend', 'Last Interaction']`
- **Categorical Variables**: `['Gender', 'Subscription Type', 'Contract Length', 'Churn']`

we can better understand customer behavior and identify potential warning signs of churn, ultimately helping businesses make data-driven decisions.

As this dataset was for machine learning, it consists of 440833 rows, I used following Python code to randomly sample 2000 out of it to avoid crashes.

```
import pandas as pd
df = pd.read_csv("customer_churn.csv")

# Sample random 2000 rows if dataset length exceeds 100,000
if len(df) > 100000:
    print(f"\nDataset contains {len(df)} rows. Sampling 2000 rows randomly...")
    df = df.sample(n=2000, random_state=42)
    print(f"Sampled dataset shape: {df.shape}")

df.to_csv("sampled_churn_data.csv", index=False, encoding="utf-8")
print("Data exported to 'sampled_churn_data.csv'")
```

Load, Clean, and Display Summary Stats

In [3]:

```python
# I developed a reusable function to load, process, and show summaries stats for a cleaned dataset.
def process_churn_data(df_path, dropna=True, dropna_threshold=0.01, col_to_drop=None, col_to_convert=None, convert_mapping={0: 'No', 1: 'Yes'}):
    """
    Cleans and summarizes a customer churn DataFrame. The data file needs to be mostly cleaned, with minor missing values.

    Parameters:
    - df_path: the path of your local data file, need to be a cleaned data file of csv, tsv, or xls.
    - dropna (bool): Whether to remove rows with NaN values below the threshold (default True).
    - dropna_threshold (float): Percentage threshold for dropping rows with NaN values (default 0.01).
    - col_to_drop (list): Columns to drop from the DataFrame if they are irrelevant (optional).
    - col_to_convert (str): Column to apply the convert_mapping conversion (optional).
    - convert_mapping (dict): Mapping to convert integer values (default {0: 'No', 1: 'Yes'}).

    Returns:
    - tuple: Cleaned DataFrame, numeric columns and categorical columns as lists, data summary table
    """

    # Step 1: Load the data based on file extension
    if df_path.endswith('.csv'):
        df = pd.read_csv(df_path)
    elif df_path.endswith('.tsv'):
        df = pd.read_csv(df_path, sep='\t')
    elif df_path.endswith('.xls') or df_path.endswith('.xlsx'):
        df = pd.read_excel(df_path)
    else:
        raise ValueError("Unsupported file format. Use .csv, .tsv, .xls, or .xlsx.")

    # Let's take a general look
    print("Let's take a quick look at our data:")
    display(df.head())

    # Step 2: Drop specific columns if provided
    if col_to_drop:
        print(f"\n1. Dropping specified columns: {col_to_drop}")
        df = df.drop(columns=col_to_drop)
        print(f"Remaining columns: {df.columns.tolist()}")

    # Step 3: Check and manage NaN values
    print("\n2. Checking and handling NaN values...")
    rows_with_nulls = df[df.isnull().any(axis=1)]
    num_rows_with_nulls = len(rows_with_nulls)
    print(f"Number of rows with NaN values: {num_rows_with_nulls}")

    if dropna and num_rows_with_nulls < dropna_threshold * len(df):
        df = df.dropna()
        print(f"Dropped {num_rows_with_nulls} rows with NaN values.")
    else:
        print(f"Retained rows with NaN values as they are more than the threshold of {dropna_threshold * 100}%.")

    # Step 4: Convert column values if specified
    if col_to_convert:
        print(f"\n3. Converting column '{col_to_convert}' values using mapping {convert_mapping}...")
        if col_to_convert in df.columns:
            df[col_to_convert] = df[col_to_convert].map(convert_mapping)
            print(f"Unique values in column '{col_to_convert}': {df[col_to_convert].unique()}")
        else:
            print(f"Column '{col_to_convert}' not found in the DataFrame.")

    # Step 5: Summarize column types, missing values, and unique entries
    numeric_cols = list(df.select_dtypes(include=[np.number]).columns)
    categorical_cols = list(df.select_dtypes(exclude=[np.number]).columns)

    # Removing col_to_convert from `categorical_cols` if it exists
    if col_to_convert and col_to_convert in categorical_cols:
        categorical_cols.remove(col_to_convert)
    print(f"\n4. The DataFrame shape is {df.shape}.")
    print(f"Numeric columns: {numeric_cols}")
    print(f"Categorical columns: {categorical_cols}")
```

```
    print("\n5. Generating detailed data summary...")

    # Summarize all columns
    data_summary = {
        col: {
            'type': str(df[col].dtype),
            'unique_values': len(df[col].unique())
        } for col in df.columns
    }

    # Extend the summary for numeric columns to include descriptive statistics
    for col in numeric_cols:
        data_summary[col].update({
            'mean': df[col].mean(),
            'median': df[col].median(),
            'std': df[col].std(),
            'min': df[col].min(),
            'max': df[col].max(),
            'range': df[col].max() - df[col].min()
        })

    # Convert summary to DataFrame for better visualization
    summary_df = pd.DataFrame(data_summary).T
    display(summary_df)

    # Step 6: Show unique values in each categorical column
    print("\n6. Unique values in categorical columns:")
    for col in categorical_cols:
        print(f"Column '{col}': {df[col].unique()}")

    return df, numeric_cols, categorical_cols, data_summary
```

In [4]:
```
# plug in our data path to function above, unpack processed df, numeric_cols and categorical_cols, and add a ";" at end) to prevent displaying returned stuff
df, numeric_cols, categorical_cols, data_summary = process_churn_data('sampled_churn_data.csv', col_to_convert='Churn', col_to_drop=['CustomerID']);
```

Let's take a quick look at our data:

	CustomerID	Age	Gender	Tenure	Usage Frequency	Support Calls	Payment Delay	Subscription Type	Contract Length	Total Spend	Last Interaction	Churn
0	71073.0	27.0	Male	14.0	28.0	3.0	16.0	Standard	Monthly	862.00	9.0	1.0
1	230192.0	40.0	Male	19.0	2.0	8.0	28.0	Standard	Monthly	620.81	21.0	1.0
2	22407.0	27.0	Female	57.0	3.0	0.0	24.0	Standard	Annual	915.00	26.0	1.0
3	290822.0	40.0	Male	21.0	14.0	0.0	11.0	Basic	Annual	592.83	9.0	0.0
4	172430.0	39.0	Male	58.0	4.0	2.0	8.0	Standard	Monthly	694.00	15.0	1.0

```
1. Dropping specified columns: ['CustomerID']
Remaining columns: ['Age', 'Gender', 'Tenure', 'Usage Frequency', 'Support Calls', 'Payment Delay', 'Subscription Type', 'Contract Length', 'Total Spend', 'Last Interaction', 'Churn']

2. Checking and handling NaN values...
Number of rows with NaN values: 0
Dropped 0 rows with NaN values.

3. Converting column 'Churn' values using mapping {0: 'No', 1: 'Yes'}...
Unique values in column 'Churn': ['Yes' 'No']

4. The DataFrame shape is (2000, 11).
Numeric columns: ['Age', 'Tenure', 'Usage Frequency', 'Support Calls', 'Payment Delay', 'Total Spend', 'Last Interaction']
Categorical columns: ['Gender', 'Subscription Type', 'Contract Length']

5. Generating detailed data summary...
```

	type	unique_values	mean	median	std	min	max	range
Age	float64	48	39.0655	39.0	12.473492	18.0	65.0	47.0
Gender	object	2	NaN	NaN	NaN	NaN	NaN	NaN
Tenure	float64	60	30.691	31.0	17.496645	1.0	60.0	59.0
Usage Frequency	float64	30	15.8315	16.0	8.696777	1.0	30.0	29.0
Support Calls	float64	11	3.612	3.0	3.108583	0.0	10.0	10.0
Payment Delay	float64	31	12.6345	12.0	8.232849	0.0	30.0	30.0
Subscription Type	object	3	NaN	NaN	NaN	NaN	NaN	NaN
Contract Length	object	3	NaN	NaN	NaN	NaN	NaN	NaN
Total Spend	float64	1638	646.06183	679.585	240.156218	103.0	999.96	896.96
Last Interaction	float64	30	14.6725	14.0	8.533385	1.0	30.0	29.0
Churn	object	2	NaN	NaN	NaN	NaN	NaN	NaN

```
6. Unique values in categorical columns:
Column 'Gender': ['Male' 'Female']
Column 'Subscription Type': ['Standard' 'Basic' 'Premium']
Column 'Contract Length': ['Monthly' 'Annual' 'Quarterly']
```

Building Individual Dashboard Components

Create a Control Panel

In [5]:

```python
def create_control_panel(var=True, group=False, range_slider_col=None, filter_col=None,
                         var_default="Variable", group_default="Group By", range_name="Range Filter", filter_name="Filter By"):
    """
    Function to create control panel elements with optional variable selector, grouping selector,
    range slider components, and a filter selector for categorical columns.

    Parameters:
    - var (bool): Whether to create a variable selector. Default is True.
    - group (bool): Whether to create a grouping selector. Default is False.
    - range_slider_col: One numerical column of choice to create a range slider. Default is None.
    - filter_col (str): Column name to create a filter for specific values (e.g., 'Churn'). Default is None.
    - var_default (str): Default name for the variable selector. Default is "Variable".
    - group_default (str): Default name for the grouping selector. Default is "Group By".
    - range_name (str): Name for the range slider. Default is "Range Filter".
    - filter_name (str): Name for the filter selector. Default is "Filter By".

    Returns:
    - select_var: The variable selector widget.
    - select_group: The grouping selector widget.
    - slider: The range slider widget (if created).
    - filter_select: The filter selector widget (if created).
    - controls (pn.Column): A Panel Column layout of the created widgets.
    """
    widgets = []

    # Create variable selector if 'var' is True
    if var:
        select_var = pn.widgets.Select(
            options=numeric_cols,  # Options are the numeric columns
            name=f"{var_default}: Choose the variable to analyze",  # Auto-generate name
            value=numeric_cols[0],  # Default to the first value in numeric_cols
            description=f"Choose the {var_default} to analyze"
        )
        widgets.append(select_var)

    # Create grouping selector if 'group' is True
    if group:
        select_group = pn.widgets.Select(
```

```
            options=categorical_cols,  # Filter categorical columns
            name=group_default,  # Default or user-specified name
            value=categorical_cols[0],  # Default to the first value in categorical_cols
            description=f"Choose the column to {group_default.lower()} by"
        )
        widgets.append(select_group)

    # Create range slider if 'range_slider_cols' is provided
    if range_slider_col:
        slider = pn.widgets.RangeSlider(
            name=f"{range_name}: {range_slider_col}",  # Auto-generate name with the column
            start=df[range_slider_col].min(),
            end=df[range_slider_col].max(),
            value=(df[range_slider_col].min(), df[range_slider_col].max()),  # Default range matches column range
            step=1,
            format='0[.]0'
        )
        widgets.append(slider)

    # Create filter selector for a specific column if 'filter_col' is provided
    if filter_col:
        filter_select = pn.widgets.Select(
            options=['Yes', 'No'],  # Unique values in the specified column
            name=f"{filter_name}: {filter_col}",  # Auto-generate name with the column
            value='Yes',  # Set default to the first value in the column
            description=f"{filter_name} column: {filter_col}"
        )
        widgets.append(filter_select)

    # Create layout with all widgets
    controls = pn.Column(
        '## Dashboard Controls',
        *widgets,  # Unpack widgets in the column
        sizing_mode='stretch_width'
    )

    return range_slider_col, filter_col, select_var, select_group, slider, filter_select, controls  # Add filter_select to the returned values
```

In [6]:
```
# Create a control panel with all options enabled
range_slider_col, filter_col, select_var, select_group, slider, filter_select, controls = create_control_panel(
    var=True,
    group=True,
    filter_col='Churn',
    range_slider_col='Total Spend', # we are particularly interested in how this variable affecting others
    var_default="Analysis Variable",
    group_default="Grouping Category",
    range_name="Filter by Range"
)

# preview the control features we just created
controls
```

Out[6]:

Dashboard Controls

Analysis Variable: Choose the variable to analyze

Age

Grouping Category

Gender

Filter by Range: Total Spend: 103 .. 1000

Filter By: Churn

Yes

Barchart

In [7]:
```
@pn.depends(select_group, slider)
def barchart(select_group, slider):

    filtered_df = df[
        (df[range_slider_col] >= slider[0]) &
        (df[range_slider_col] <= slider[1])
    ]

    # create barchart
    bar_chart = filtered_df.groupby([select_group, filter_col]).size().unstack().hvplot.bar(
        x=select_group,
        y=['Yes', 'No'],
        stacked=True,
        title=f'{filter_col} by {select_group}',
        xlabel='select_group',
        ylabel='Count',
        legend='top'
    )

    return bar_chart

barchart_layout = pn.Column(
    pn.pane.Markdown("### Interactive Bar Chart (Demo Only, Do Not Click)"),
    select_group,
    slider,
    pn.panel(barchart),
    sizing_mode="stretch_both"
)

# Display the dashboard
barchart_layout.servable()
```

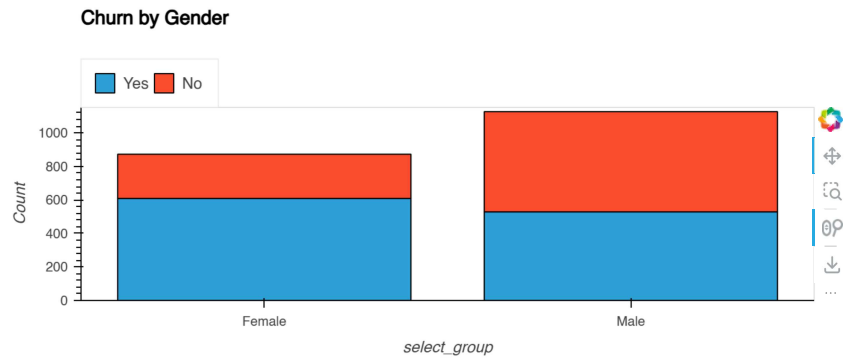
Out[7]:

Interactive Bar Chart (Demo Only, Do Not Click)

Grouping Category ⓘ

Gender ▾

Filter by Range: Total Spend: 103 .. 1000



Histogram

In [8]:
```
# @pn.depends tells Panel which widgets should trigger updates
@pn.depends(select_var, slider, filter_select)
def histogram_plot(select_var, slider, filter_select):
    """
    Creates an interactive histogram for selected variable with density curves.

    Args:
```

```
        select_var (str): The variable to plot
        select_group (str): The grouping variable

    Returns:
        hvplot: Interactive histogram plot
    """

    # filter data based on parameters
    selected_df = df[df[filter_col] == filter_select]

    filtered_df = selected_df[
        (selected_df[range_slider_col] >= slider[0]) &
        (selected_df[range_slider_col] <= slider[1])
    ]

    # add density curve
    x_min = filtered_df[select_var].min()
    x_max = filtered_df[select_var].max()
    sw = np.linspace(x_min, x_max, 1000)
    fit = stats.norm.pdf(sw, np.mean(filtered_df[select_var]), np.std(filtered_df[select_var]))
    bin_width = (x_max - x_min) / 20
    fit_scaled = fit * len(filtered_df) * bin_width

    density_curve = hv.Curve((sw, fit_scaled)).opts(
    line_width=2,
    color='red'
)

    # Create the plot with some customization
    histogram = filtered_df.hvplot.hist(
        y=select_var,
        bins=20,
        height=300,
        alpha=0.5,
        title=f'Histogram for {select_var}',
        xlabel=select_var,
        ylabel='Count',
        **{'responsive': True,
           'legend_position': 'right'}
    )

    return density_curve * histogram

hist_layout = pn.Column(
    pn.pane.Markdown("### Interactive Historgram (Demo Only, Do Not Click)"),
    select_var, select_group, slider, filter_select,
    pn.panel(histogram_plot),
    sizing_mode="stretch_both"
)

# Display the dashboard
hist_layout.servable()
```

Out[8]:

Interactive Historgram (Demo Only, Do Not Click)

Analysis Variable: Choose the variable to analyze

Age

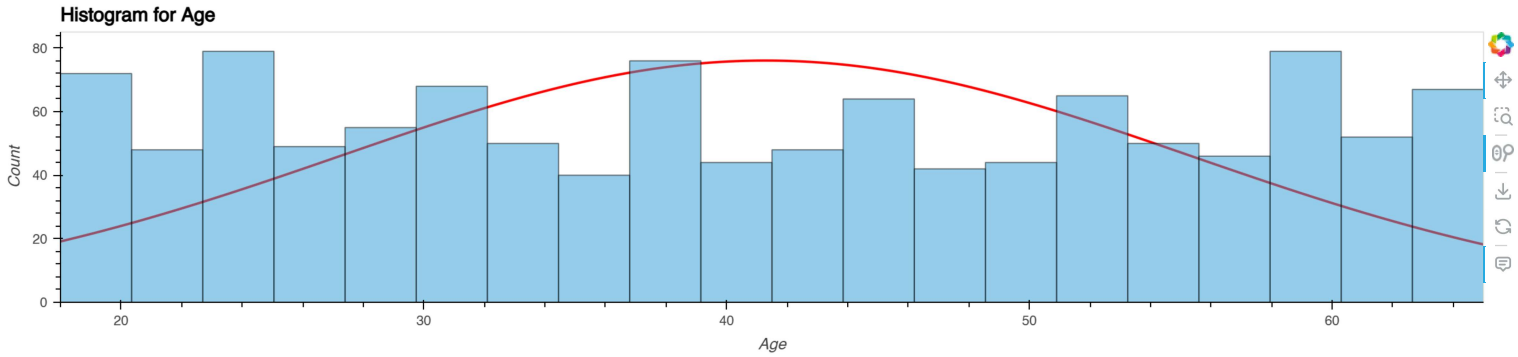
Grouping Category

Gender

Filter by Range: Total Spend: 103 .. 1000

Filter By: Churn

Yes



Boxplot

In [9]:

```
@pn.depends(select_var, select_group, slider, filter_select)
def box_plot(select_var, select_group, slider, filter_select):
    """Creates a box plot for selected variables and range, if provided. It also includes hover tooltips for more interactivity.

    Args:
        select_var (str): Variable to plot on y-axis
        select_group (str): Grouping variable for x-axis

    Returns:
        hvplot: Interactive box plot
    """
    # First, let's filter our data

    # filter data based on parameters
    selected_df = df[df[filter_col] == filter_select]

    filtered_df = selected_df[
        (selected_df[range_slider_col] >= slider[0]) &
        (selected_df[range_slider_col] <= slider[1])
    ].copy() # Create a copy to avoid SettingWithCopyWarning


    # get statistics for hover tooltips
    stats = {
        group: {
            'median': filtered_df[filtered_df[select_group]==group][select_var].median(),
            'mean': filtered_df[filtered_df[select_group]==group][select_var].mean(),
            'std': filtered_df[filtered_df[select_group]==group][select_var].std()
        } for group in filtered_df[select_group].unique()
    }

    # Create the box plot
    plot = filtered_df.hvplot.box(
```

```
        y=select_var,
        by=select_group,
        height=300,
        whisker_color='black',
        title=f'Boxplot for {select_var} by {select_group}',

        # Customize appearance
        box_alpha=0.7,
        outlier_alpha=0.7,
        width=400,
        legend='top',

        # Add statistical hover texts
        tools=['hover'],
        tooltips=[
            ('Group', '@{' + select_group + '}'),
            ('Value', '@{' + select_var + '}{0.00}'),
            ('Count', '@count'),
            ('Median', '@median{0.00}')
        ]
    )

    return plot

boxplot_layout = pn.Column(
    pn.pane.Markdown("### Interactive Boxplot (Demo Only, Do Not Click)"),
    select_var, select_group, slider, filter_select,
    pn.panel(box_plot),
    sizing_mode="stretch_both"
)

# Display the dashboard
boxplot_layout.servable()
```

Out[9]:

Interactive Boxplot (Demo Only, Do Not Click)

Analysis Variable: Choose the variable to analyze

Age

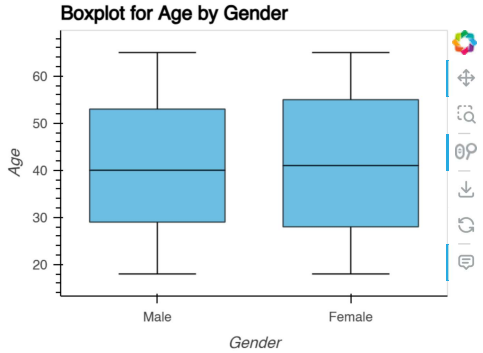
Grouping Category

Gender

Filter by Range: Total Spend: 103 .. 1000

Filter By: Churn

Yes



Scatterplot

In [10]:

```python
@pn.depends(select_var, slider, filter_select, select_group)
def create_scatter(x_var, slider, filter_select, group_var):
    """
    Creates an interactive scatter plot with points grouped by a specified column.

    Args:
        x_var (str): The variable to use on the x-axis.
        slider (RangeSlider): RangeSlider object to filter data by a numeric range.
        filter_select (str): Column value used to filter rows (e.g., 'Yes' for column 'Churn').
        group_var (str): Column name to group data points by (e.g., 'Gender').

    Returns:
        hvPlot: An interactive scatter plot, grouped by the specified column with colors, with hover tools and interactivity.
    """

    # Filter data based on parameters
    selected_df = df[df[filter_col] == filter_select]

    filtered_df = selected_df[
        (selected_df[range_slider_col] >= slider[0]) &
        (selected_df[range_slider_col] <= slider[1])
    ]

    # Determine the y-variable
    y_var = range_slider_col if x_var != range_slider_col else 'hp'

    # Identify unique groups
    groups = filtered_df[group_var].unique()

    combined = None

    for i, g in enumerate(groups):
        group_data = filtered_df[filtered_df[group_var] == g]

        # Introduce jitter to reduce overlapping points
        group_data[x_var] += np.random.uniform(-0.5, 0.5, size=len(group_data))
        group_data[y_var] += np.random.uniform(-0.5, 0.5, size=len(group_data))

        # Create scatter plot for this group with transparency
        scatter = group_data.hvplot.scatter(
            x=x_var,
            y=y_var,
            alpha=0.6,
            label=str(g),
            size=5
        )

        # Combine with previous groups
        if combined is None:
            combined = scatter
        else:
            combined = combined * scatter

    # Add options to the combined plot
    plot = combined.opts(
        width=600,
        height=400,
        title=f'Relationship between {x_var} and {y_var}\n(grouped by {group_var})',
        tools=['hover', 'box_zoom', 'reset'],
        show_grid=True,
        toolbar='above'
    )

    return plot

scatter_layout = pn.Column(
    pn.pane.Markdown("### Interactive Scartterplot (Demo Only, Do Not Click)"),
    select_var, select_group, slider, filter_select,
    pn.panel(create_scatter),
```

```
    sizing_mode="stretch_both"
)

# Display the dashboard
scatter_layout.servable()
```

Out[10]:

Interactive Scartterplot (Demo Only, Do Not Click)

Analysis Variable: Choose the variable to analyze

Age

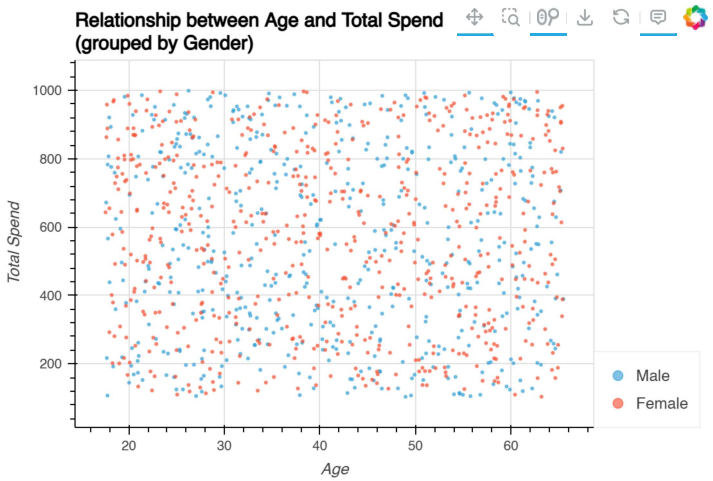
Grouping Category

Gender

Filter by Range: Total Spend: 103 .. 1000

Filter By: Churn

Yes



Build a Tabbed Dashboard by Combining All Visualizations

In [11]:

```
pn.extension()

def create_dashboard(widgets, plots):
    # Extract widgets
    select_var, select_group, slider, filter_select = widgets

    # Create separate control layouts for each tab
    # General Tab: Only select_group and slider
    general_controls = pn.Column(
        select_group,
        slider,
        sizing_mode="stretch_width"
    )

    # Distributions and Correlations Tabs: All widgets
    other_controls = pn.Column(
        *widgets,  # Include all widgets
        sizing_mode="stretch_width"
    )

    # Create tabs
```

```
    tabs = pn.Tabs(
        ('Categorical', pn.Column(
            general_controls,  # Only select_group and slider
            pn.Row(plots['barchart'], sizing_mode='stretch_both'),
            sizing_mode='stretch_both'
        )),
        ('Numerial', pn.Column(
            other_controls,  # All widgets
            pn.Row(plots['boxplot'], plots['histogram'], sizing_mode='stretch_both'),
            sizing_mode='stretch_both'
        )),
        ('Correlations', pn.Row(
            other_controls,  # All widgets
            plots['scatter'],
            sizing_mode='stretch_both'
        )),
        ('Statistics', pn.Column(
            # No widgets for stats tab
            plots['stats'],
            sizing_mode='stretch_both'
        )),
        sizing_mode='stretch_both'
    )

    main_layout = pn.Column(tabs, sizing_mode='stretch_both').servable()

    template = pn.template.VanillaTemplate(
        title="Interactive EDA Dashboard",
        sidebar=[],
        main=[main_layout],
    )

    return template

# Initialize and display the dashboard
dashboard = create_dashboard(
    widgets=[select_var, select_group, slider, filter_select],
    plots={
        'barchart': barchart,
        'histogram': histogram_plot,
        'boxplot': box_plot,
        'scatter': create_scatter,
        'stats': pd.DataFrame(data_summary)
    }
)

dashboard.show()
```

```
Launching server at http://localhost:46713
```

Out[11]: <panel.io.server.Server at 0x7c7d682d71c0>