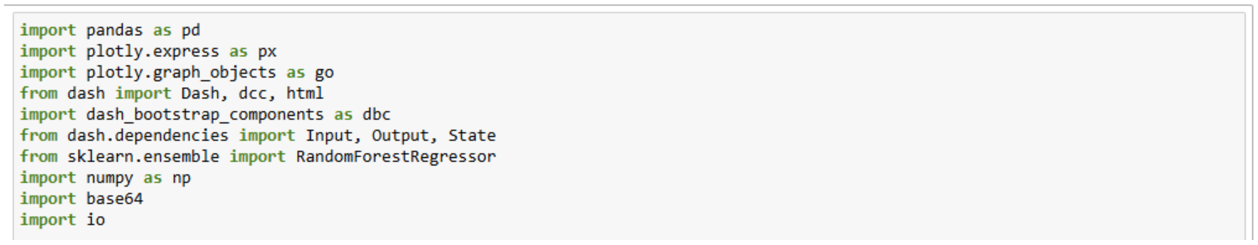
Make Sure you have Jupyter Notebook installed

Step 1: Importing Necessary Libraries

To ensure all necessary packages are installed within the Jupyter environment, the following command can be executed in a Jupyter Notebook cell:

'!pip install pandas plotly dash dash-bootstrap-components scikit-learn numpy'

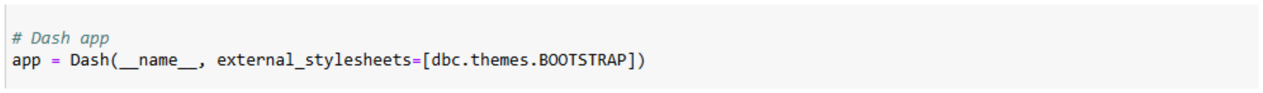
This command will install all the required libraries to run the application smoothly.



```
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from dash import Dash, dcc, html
import dash_bootstrap_components as dbc
from dash.dependencies import Input, Output, State
from sklearn.ensemble import RandomForestRegressor
import numpy as np
import base64
import io
```

- **Pandas (`pd`)**: A data loading, manipulation and analysis library.
- **Plotly (`px`, `go`)**: A graphing library to create interactive plots and visualizations.
- **Dash (`Dash`, `dcc`, `html`)**: A web application library to build interactive web applications in Python. It provides the components for the layout and interactivity of the application.
- **Dash Bootstrap Components (`dbc`)**: A library with Dash that helps to style and organize the layout of the application.
- **Dash Dependencies (`Input`, `Output`, `State`)**: These are used to define the interactivity in the app, allowing components to respond to user inputs.
- **Scikit-Learn (`RandomForestRegressor`)**: A machine learning library that was used to create a model for predicting data planes.
- **Numpy (`np`)**: A numerical computing library, used here for handling arrays and performing mathematical operations needed for the regression model.
- **Base64**: Used for encoding and decoding the content of the uploaded Excel file.
- **IO (`io`)**: Provides tools for handling input and output operations, specifically to read the in-memory file from the uploaded data.

Step 2: Setting Up the Dash Application



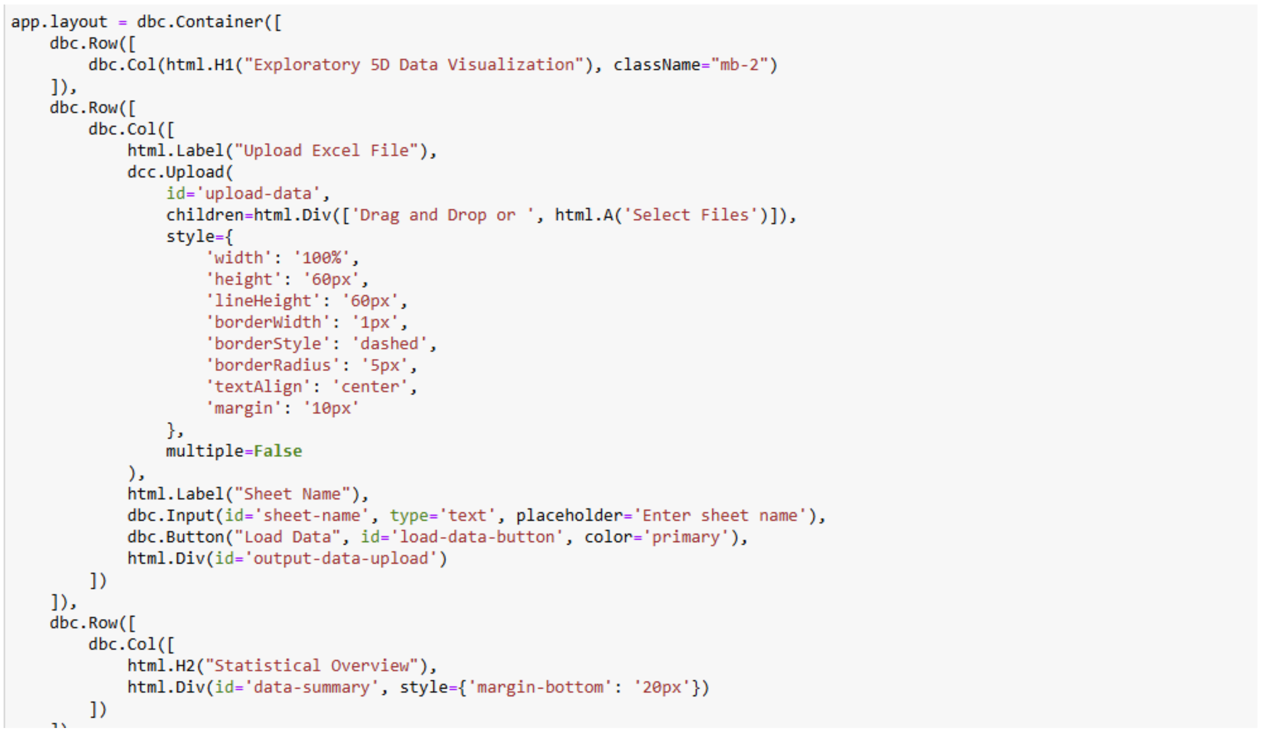
```
# Dash app
app = Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])
```

In this step, the Dash application is initialized

- **`Dash(__name__)`**: This command initializes an instance of the Dash application. This line creates a web application.

- **`external_stylesheets=[dbc.themes.BOOTSTRAP]`**: Here, the application is configured to use a predefined design style called Bootstrap. Bootstrap provides a set of design guidelines that help the application look clean, organized, and visually appealing.

Step 3: Designing the Application Layout (Part 1)



```python
app.layout = dbc.Container([
    dbc.Row([
        dbc.Col(html.H1("Exploratory 5D Data Visualization"), className="mb-2")
    ]),
    dbc.Row([
        dbc.Col([
            html.Label("Upload Excel File"),
            dcc.Upload(
                id='upload-data',
                children=html.Div(['Drag and Drop or ', html.A('Select Files')]),
                style={
                    'width': '100%',
                    'height': '60px',
                    'lineHeight': '60px',
                    'borderWidth': '1px',
                    'borderStyle': 'dashed',
                    'borderRadius': '5px',
                    'textAlign': 'center',
                    'margin': '10px'
                },
                multiple=False
            ),
            html.Label("Sheet Name"),
            dbc.Input(id='sheet-name', type='text', placeholder='Enter sheet name'),
            dbc.Button("Load Data", id='load-data-button', color='primary'),
            html.Div(id='output-data-upload')
        ])
    ]),
    dbc.Row([
        dbc.Col([
            html.H2("Statistical Overview"),
            html.Div(id='data-summary', style={'margin-bottom': '20px'})
        ])
```

- **Container**:
    - The entire layout is wrapped inside a ‘`dbc.Container`’, which acts like a main box holding all the elements together. This container ensures that the content is centered and responsive, adjusting to different screen sizes.

- **First Row: Application Title**:
    - The first row* (`dbc.Row`) contains a single column** (`dbc.Col`) that spans the entire width of the row. Inside this column, there is a header (`html.H1`) displaying the title of the app "Exploratory 5D Data Visualization".
    - The title is styled with a small bottom margin (`className="mb-2"`) to add space below it, making the layout more visually appealing.

*A **Row** is a horizontal section that stretches across the width of the container. Rows help organize content horizontally, so different sections of the app can be placed on top of one another

** A **Column** is a vertical section within a row. Columns divide the row into sections where content is placed.

- **Second Row: File Upload and Data Input**:

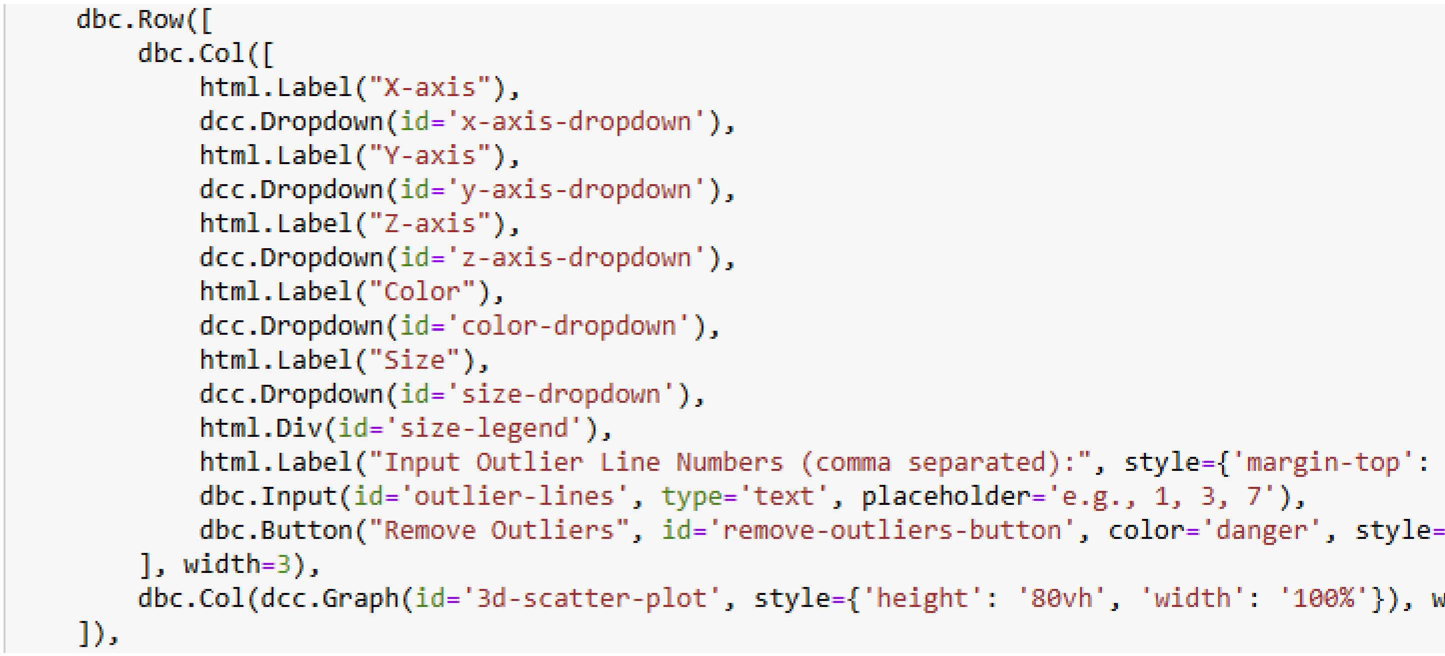
  - The second row is where users interact with the app by uploading their data and specifying additional details.
  - **Upload Excel File**:
    - A label (`html.Label`) prompts the user to upload an Excel file.
    - The '`dcc.Upload`' component provides the functionality for the file upload. Users can either drag and drop the file into the designated area or click to select a file from their computer. This area is styled to have a dashed border, rounded corners, and centered text, making it easy to use and visually clear.
  - **Sheet Name Input**:
    - Below the file upload section, there is an input box (`dbc.Input`) where users can enter the name of the specific sheet from the Excel file that they want to load. This input helps to specify which part of the data to use if the file contains multiple sheets.
  - **Load Data Button**:
    - A button labeled "Load Data" (`dbc.Button`) is provided to trigger the loading of the selected file and sheet. This button is styled with a primary color to make it stand out.
  - **Output Area**:
    - An empty '`html.Div`'* is added to the layout to serve as a placeholder where messages or other content will appear after the data is loaded. This section will automatically update and show new information based on what the user does, such as when they upload a file or interact with other parts of the app.

    *A `Div` (short for "division") is like a box in the web page where different types of content can be put.

- **Third Row: Statistical Overview**:

  - The third row is designed to present a statistical summary of the data once it is loaded.
  - It contains a header (`html.H2`) titled "Statistical Overview" to indicate what this section is about.
  - Below the header, a placeholder `html.Div` is included, which will later display the summary of the data. The `style={'margin-bottom': '20px'}` adds space below this section to ensure a clean and organized look.

Step 3: Designing the Application La



```
    dbc.Row([
        dbc.Col([
            html.Label("X-axis"),
            dcc.Dropdown(id='x-axis-dropdown'),
            html.Label("Y-axis"),
            dcc.Dropdown(id='y-axis-dropdown'),
            html.Label("Z-axis"),
            dcc.Dropdown(id='z-axis-dropdown'),
            html.Label("Color"),
            dcc.Dropdown(id='color-dropdown'),
            html.Label("Size"),
            dcc.Dropdown(id='size-dropdown'),
            html.Div(id='size-legend'),
            html.Label("Input Outlier Line Numbers (comma separated):", style={'margin-top':
            dbc.Input(id='outlier-lines', type='text', placeholder='e.g., 1, 3, 7'),
            dbc.Button("Remove Outliers", id='remove-outliers-button', color='danger', style=
        ], width=3),
        dbc.Col(dcc.Graph(id='3d-scatter-plot', style={'height': '80vh', 'width': '100%'}), w
    ]),
```

yout (Part 2)

- **Row Structure**:
    - Same thing with the row and column, but this time the first column is narrower (takes up 3 units of width as shown in the figure above), and the second column is wider (takes up 9 units of width), making the entire row a 12-unit grid, which is a common design convention in web layouts.

- **First Column (User Controls)**:
    - The first column is packed with controls that allow users to interact with the data:
        - **Dropdowns**:
            - There are several dropdown menus (`dcc.Dropdown`) labeled for different axes (X-axis, Y-axis, Z-axis) and for visual properties like color and size. These dropdowns will be populated with options once the data is loaded, allowing users to choose which data variables they want to visualize on each axis or through color and size.
        - **Outlier Input**:
            - A label and an input box (`dbc.Input`) allow users to enter specific rows (lines) in the data that they want to treat as outliers and remove from the visualization. This is where users can clean up their data by specifying which points should be excluded.
        - **Remove Outliers Button**:
            - A button labeled "Remove Outliers" (`dbc.Button`) is provided for users to apply the outlier removal. It is styled with a red color (`color='danger'`) to indicate that this action will alter the data displayed.

- **Second Column (3D Scatter Plot)**:
  - The second column is dedicated to displaying the main data visualization:
    - **3D Scatter Plot**:
      - A large graph component (`dcc.Graph`) is placed here, taking up most of the screen width. It is configured to display a 3D scatter plot that will update dynamically based on the user's selections in the first column.
      - The height and width of the graph are set (`style={'height': '80vh', 'width': '100%'}`) to ensure it takes up a significant portion of the screen, making it easy for users to interact with and interpret the data.

Step 4: Handling Data Upload and Initialization

```
df = pd.DataFrame()

@app.callback(
    Output('output-data-upload', 'children'),
    Output('x-axis-dropdown', 'options'),
    Output('y-axis-dropdown', 'options'),
    Output('z-axis-dropdown', 'options'),
    Output('color-dropdown', 'options'),
    Output('size-dropdown', 'options'),
    Output('data-summary', 'children'),
    Input('load-data-button', 'n_clicks'),
    State('upload-data', 'contents'),
    State('sheet-name', 'value')
)
def load_data(n_clicks, contents, sheet_name):
    if n_clicks is None or contents is None:
        return '', [], [], [], [], [], ''

    content_type, content_string = contents.split(',')
    decoded = base64.b64decode(content_string)
    global df
    df = pd.read_excel(io.BytesIO(decoded), sheet_name=sheet_name)

    options = [{'label': col, 'value': col} for col in df.columns]
    summary = df.describe().reset_index().to_dict('records')

    table_header = [html.Thead(html.Tr([html.Th(col) for col in df.describe().reset_index().c
    table_body = [html.Tbody([html.Tr([html.Td(value) for value in row.values()]) for row in

    summary_table = dbc.Table(table_header + table_body, bordered=True, striped=True, hover=T

    return 'Data Loaded', options, options, options, options, options, summary_table
```

In this step, the code handles the process of uploading an Excel file, reading the data, and preparing it for visualization. The function used here is called a '**callback**' in Dash, which means it automatically updates certain parts of the app when specific actions occur.

- **Global DataFrame Initialization**:

  - `'df = pd.DataFrame()'`:
    - A global Data Frame is created as an empty container to hold the data from the uploaded Excel file. This Data Frame (df) will be filled and used throughout the application once the data is loaded.

- **Defining the Callback Function**:

  - **Callback Decorator**:
    - `'@app.callback(...)'`:
      - This line defines a callback function that automatically updates certain parts of the app when specific inputs change. In this case, it updates several dropdown menus, a data summary table, and the upload status message (after loading the data).
    - The **Outputs** of this callback are the elements in the app that will be updated, like the dropdown options and the summary table.
    - The **Inputs** are the triggers for the next function, such as when the "Load Data" button is clicked.
    - **State** elements are additional inputs that don't directly trigger the function but provide necessary information, like the contents of the uploaded file and the sheet name.

- **Loading the Data**:

  - `'def load_data(n_clicks, contents, sheet_name)':`:
    - This function runs whenever the "Load Data" button is clicked (from the callback input).
    - If no file is uploaded (`contents is None`) or the button hasn't been clicked, it returns empty values to prevent any further actions.

- **Processing the Uploaded File**:

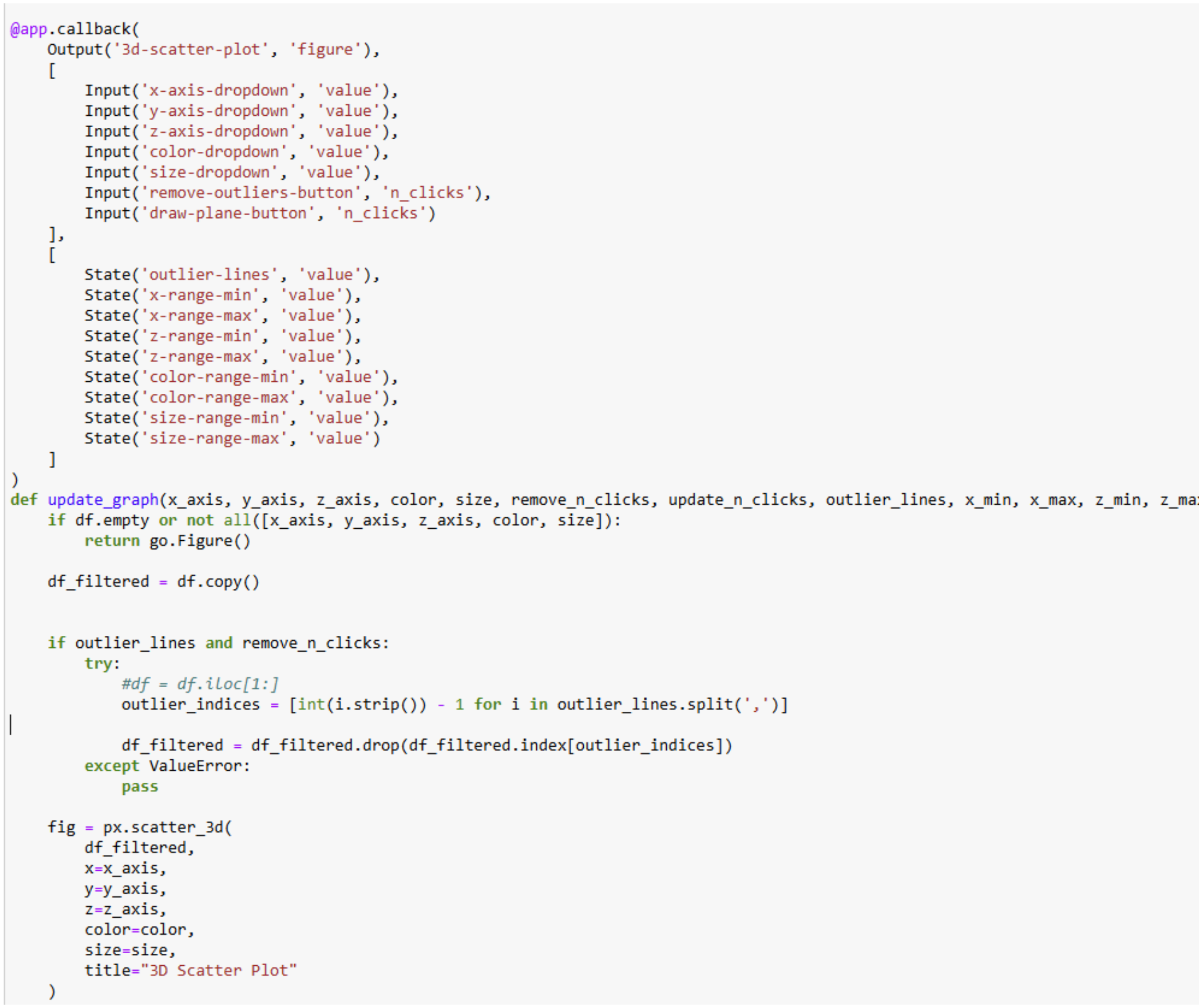
  - The uploaded file's content is decoded from a 'base64' format, which is a way to safely transmit the file data.
  - The Excel file is read into the global `df` using '`pd.read_excel'`, which allows for further manipulation and analysis within the app.

- **Updating Dropdown Options**:

  - The columns of the df are extracted and formatted into a list of options that populate the dropdown menus. Each column in the data represents a possible choice for visualization axes or attributes like color and size.

- **Creating a Data Summary Table**:
  - A summary of the data is generated using ‘`df.describe()`’, which provides the statistical information such as mean, min, max etc.
  - This summary is formatted into a table (`summary_table`) that is displayed in the app, allowing users to quickly see an overview of their data.
- **Returning Updated Content**:
  - The function returns the message "Data Loaded", the updated dropdown options, and the summary table to be displayed in the app. This ensures that the app dynamically reflects the uploaded data and provides users with the tools to interact with it.

Step 5: Updating the 3D Scatter Plot (Part 1)



```
@app.callback(
    Output('3d-scatter-plot', 'figure'),
    [
        Input('x-axis-dropdown', 'value'),
        Input('y-axis-dropdown', 'value'),
        Input('z-axis-dropdown', 'value'),
        Input('color-dropdown', 'value'),
        Input('size-dropdown', 'value'),
        Input('remove-outliers-button', 'n_clicks'),
        Input('draw-plane-button', 'n_clicks')
    ],
    [
        State('outlier-lines', 'value'),
        State('x-range-min', 'value'),
        State('x-range-max', 'value'),
        State('z-range-min', 'value'),
        State('z-range-max', 'value'),
        State('color-range-min', 'value'),
        State('color-range-max', 'value'),
        State('size-range-min', 'value'),
        State('size-range-max', 'value')
    ]
)
def update_graph(x_axis, y_axis, z_axis, color, size, remove_n_clicks, update_n_clicks, outlier_lines, x_min, x_max, z_min, z_ma
    if df.empty or not all([x_axis, y_axis, z_axis, color, size]):
        return go.Figure()

    df_filtered = df.copy()


    if outlier_lines and remove_n_clicks:
        try:
            #df = df.iloc[1:]
            outlier_indices = [int(i.strip()) - 1 for i in outlier_lines.split(',')]

            df_filtered = df_filtered.drop(df_filtered.index[outlier_indices])
        except ValueError:
            pass

    fig = px.scatter_3d(
        df_filtered,
        x=x_axis,
        y=y_axis,
        z=z_axis,
        color=color,
        size=size,
        title="3D Scatter Plot"
    )
```

The function responsible for updating the 3D scatter plot is defined under a Dash callback. This callback is triggered by various user interactions, such as selecting axes from dropdowns, removing outliers, or adjusting range values (the callback).

- **Initial Check for Data and Selections**:

    - The function begins by checking if the global `df` is empty or if any of the key axes (`x_axis`, `y_axis`, `z_axis`, `color`, `size`) have not been selected. If either condition is true, the function returns an empty plot (`go.Figure()`). This ensures the plot is only created when the necessary data and selections are available.

- **Copying the df**:

    - A copy of the `df` is made (`df_filtered = df.copy()`). This allows the function to modify the data (like removing outliers) without affecting the original data, preserving data integrity.

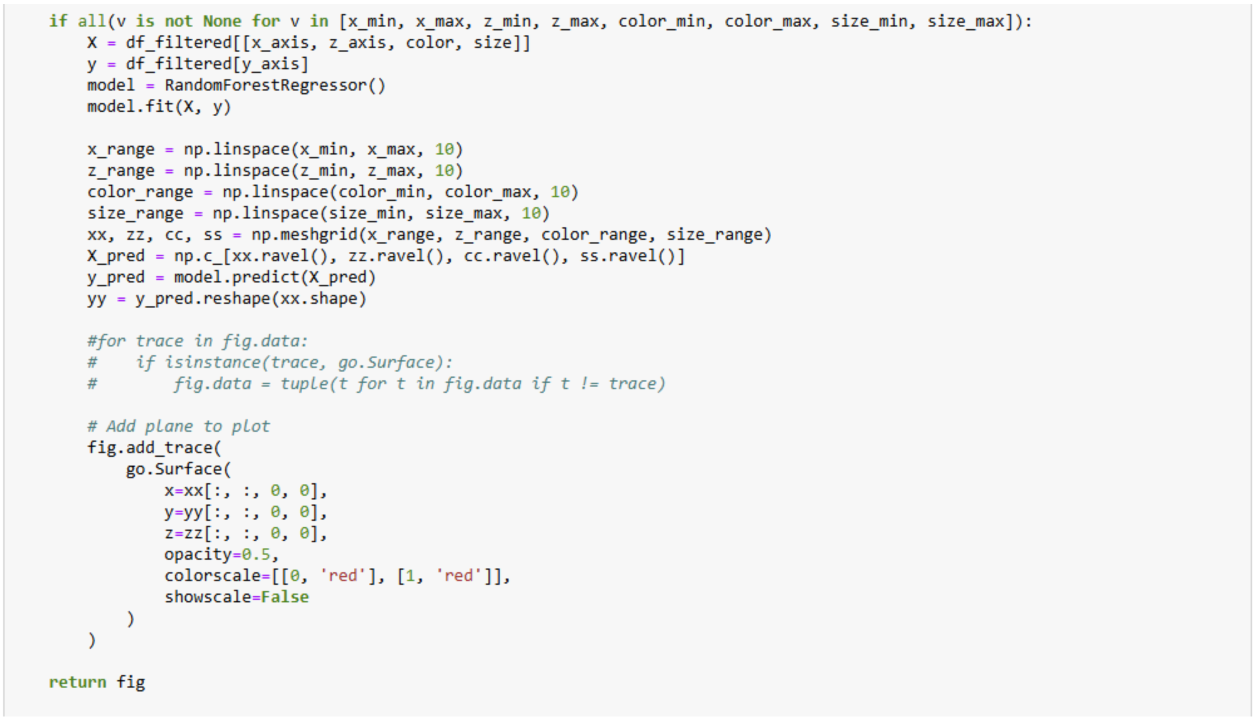
- **Removing Outliers**:

    - If the user has provided specific lines to remove as outliers and clicked the "Remove Outliers" button, the function attempts to remove them from the copied `df`.
    - The outlier lines are expected to be entered as a comma-separated list of numbers and used to drop the corresponding rows from the `df` (e.g. '1,3,9' input will remove lines 13 and 9).
    - If the input for outlier lines is invalid (e.g., not a number), the function catches the error and continues without making changes.

- **Creating the 3D Scatter Plot**:

    - A 3D scatter plot is generated using '`plotly.express.scatter_3d`'. This function takes the filtered `df` and maps the selected columns to the x, y, and z axes, as well as to color and size dimensions.
    - The plot is titled "3D Scatter Plot", and the axes are labeled with the names of the selected columns.

Step 5: Updating the 3D Scatter Plot (Part 2)



```python
if all(v is not None for v in [x_min, x_max, z_min, z_max, color_min, color_max, size_min, size_max]):
    X = df_filtered[[x_axis, z_axis, color, size]]
    y = df_filtered[y_axis]
    model = RandomForestRegressor()
    model.fit(X, y)

    x_range = np.linspace(x_min, x_max, 10)
    z_range = np.linspace(z_min, z_max, 10)
    color_range = np.linspace(color_min, color_max, 10)
    size_range = np.linspace(size_min, size_max, 10)
    xx, zz, cc, ss = np.meshgrid(x_range, z_range, color_range, size_range)
    X_pred = np.c_[xx.ravel(), zz.ravel(), cc.ravel(), ss.ravel()]
    y_pred = model.predict(X_pred)
    yy = y_pred.reshape(xx.shape)

    #for trace in fig.data:
    #    if isinstance(trace, go.Surface):
    #        fig.data = tuple(t for t in fig.data if t != trace)

    # Add plane to plot
    fig.add_trace(
        go.Surface(
            x=xx[:, :, 0, 0],
            y=yy[:, :, 0, 0],
            z=zz[:, :, 0, 0],
            opacity=0.5,
            colorscale=[[0, 'red'], [1, 'red']],
            showscale=False
        )
    )

return fig
```

In the second part of the ‘`update_graph`’ function, the code focuses on adding an **optional** prediction plane to the 3D scatter plot.

- **Check for Range Values**:

  - The code first checks if all the range values (`x_min`, `x_max`, `z_min`, `z_max`, `color_min`, `color_max`, `size_min`, `size_max`) are provided by the user. If all these values are available, the function proceeds to create the prediction plane.
  - This check ensures that the prediction plane is only added when complete information is provided, preventing incomplete or misleading visualizations.

- **Prepare the Data for Prediction**:

  - **Input Data (`X`) and Target (`y`)**:
    - o The selected columns for the x-axis, z-axis, color, and size are extracted from ‘`df_filtered`’ and stored in `X`, which represents the input features for the model.
    - o The selected y-axis variable is stored in `y`, which serves as the target variable that the model will predict.
  - **Model Fitting**:
    - o A ‘`RandomForestRegressor`’ model is initialized and fitted on the input data `X` and `y`. This model is used to predict the y-values for the plane based on the specified ranges.

- **Create a Grid of Values for Prediction**:

  - **Generate Ranges**:

    - For each dimension (x, z, color, size), the code generates a range of values using ‘`np.linspace`’. This function creates evenly spaced values between the specified minimum and maximum limits.
- **Meshgrid Creation**:
    - The ‘`np.meshgrid`’ function is used to create a grid of values (`xx`, `zz`, `cc`, `ss`) that cover all possible combinations within the specified ranges. This grid represents the plane in the 3D space.
- **Predicting y-values**:
    - The model predicts y-values (`y_pred`) for each combination of x, z, color, and size values in the grid. These predictions are reshaped to match the shape of the grid, allowing the plane to be plotted in the correct format.

Check the thesis text to have more details how the plane was drawn below figure 31 in section 5.3.2

- **Add the Prediction Plane to the Plot**:

    - The new plane is added to the plot using ‘`go.Surface`’. This surface is colored red and is semi-transparent (`opacity=0.5`) so that it does not obscure the scatter plot but still provides a clear visualization of the predicted trend.
    - The plane is positioned in the 3D space according to the x, z, and predicted y-values, giving users a visual representation of the model's predictions.

- **Return the Updated Plot**:

    - Finally, the function returns the updated figure (`fig`) with the scatter plot and the prediction plane, which is then displayed in the app.

Step 6: Updating the Size Legend

```
@app.callback(
    Output('size-legend', 'children'),
    [Input('size-dropdown', 'value')]
)
def update_size_legend(size):
    if df.empty or size is None:
        return ""

    min_size = df[size].min()
    max_size = df[size].max()
    return html.Div([
        html.Hr(),
        html.Label(f"Minimum size variable value: {min_size}"),
        html.Br(),
        html.Label(f"Maximum size variable value: {max_size}")
    ])
```

This part of the code is responsible for displaying the size legend underneath the dropdown menus. The size legend provides users with an idea of the minimum and maximum values of the size variable, helping them get a feel for the data points in the scatter plot. Since the app includes a legend for colors but not for size, this addition ensures that users can also understand how the size of each point corresponds to the underlying data.

- **Initial Check**:
    - The function starts by checking if the df is empty or if no size variable is selected (`size is None`). If either of these conditions is true, the function returns an empty string, meaning nothing will be displayed in the space.

- **Calculate Size Range**:
    - If the df is not empty and a size variable is selected, the function calculates the minimum (`min_size`) and maximum (`max_size`) values of the selected size variable from the df.
    - These values represent the smallest and largest values in the selected column, which will correspond to the smallest and largest points in the scatter plot.

- **Display the Size Range**:
    - The function returns an HTML div that contains:
        - A horizontal rule (`html.Hr()`) to separate the legend from other content visually (the dropdowns).
        - Two labels displaying the minimum and maximum size values. These labels provide users with a clear understanding of how the size of the points in the plot relates to the data values.
    - The use of line breaks (`html.Br()`) ensures that the labels are displayed on separate lines, making the legend easy to read.

Step 7: Running the Application

```
if __name__ == '__main__':
    app.run_server(debug=True)

#http://127.0.0.1:8050/
```

In this (final) part of the code, the application is set up to run and can be accessible through a web browser. This final step ensures that the web application is live and ready to be used, with the specified URL (next to the '#' which is: http://127.0.0.1:8050/) providing access to the interface.