Data Science 677 Final Project

Pagadala Shruti Krishna
U32508499

The aim of the project is to analyze the cybersecurity dataset file to obtain the results and analyze- the heatmaps, attack signatures and perform classification models.

A brief description of the code: -

1. **Importing Libraries**:
    - **NumPy** as **np**: A popular library for numerical computations in Python.
    - **pandas** as **pd**: A library used for data manipulation and analysis.
    - **seaborn** as **sns**: A statistical data visualization library.
    - **matplotlib.pyplot** as **plt**: A plotting library for creating static, interactive, and animated visualizations in Python.
2. **Importing os Library**: The **os** module in Python provides functions for interacting with the operating system. **os** is used here to navigate and interact with the file system.
3. **Setting a Directory Path**:
    - **directory_path** is set to the string **'/Users/shrutikrishnapagadala/Downloads/'**. This is likely the path to a Downloads folder for a user named Shrutikrishna Pagadala.
4. **Navigating the File System**:
    - **os.walk(directory_path)**: This function generates the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at the directory top (including top itself), it yields a 3-tuple **(dirpath, dirnames, filenames)**.
        - **dirpath** is a string, the path to the directory.
        - **dirnames** is a list of the names of the subdirectories in **dirpath** (excluding '.' and '..').
        - **filenames** is a list of the names of the non-directory files in **dirpath**.
5. **Printing File Paths**:
    - The **for** loops iterate over each directory and file in the **directory_path**.
    - **os.path.join(dirname, filename)** constructs the full path to each file by joining the directory name and the file name.
    - **print()** then outputs the full path of each file within the specified Downloads directory to the console.

Essentially, this script is used to list all the files in the specified Downloads directory and its subdirectories, printing out the full paths of these files. It does not modify any files or directories; it merely lists them. The imports for **numpy**, **pandas**, **seaborn**, and **matplotlib** are not used in the script and are redundant for the functionality shown.

- **Reading a CSV File into a DataFrame**:
    - **pd.read_csv('/Users/shrutikrishnapagadala/Downloads/cybersecurity_attacks.csv')**: This function is part of the pandas library, where **pd** is the common alias for pandas. It reads the contents of a CSV file located at the specified path and loads it into a pandas DataFrame.

- The file path **'/Users/shrutikrishnapagadala/Downloads/cybersecurity_attacks.csv'** suggests that the CSV file named **cybersecurity_attacks.csv** is in the Downloads folder of a user named Shrutikrishna Pagadala.
- The result of this operation (**df**) is a DataFrame, which is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). This DataFrame (**df**) now contains the data from the **cybersecurity_attacks.csv** file and can be used for data analysis, manipulation, and visualization.

- The **df.info()** function in this context is used to display a concise summary of a DataFrame.

- The **df.head()** function in this context is used to view the first few rows of a pandas DataFrame

```
print(df.isnull().sum())
```

The **print(df.isnull().sum())** code performs a two-step operation to identify and sum up the number of missing (null) values in each column of the DataFrame **df**. Here's a breakdown of its functionality:

1. **Identifying Null Values**:
   - **df.isnull()**: This function is applied to the DataFrame **df**. It returns a new DataFrame where each cell is either **True** or **False**, depending on whether the corresponding cell in **df** is null (missing data) or not. Essentially, for each cell in **df**, if the value is **NaN** (Not a Number) or otherwise missing, **isnull()** marks it as **True**; otherwise, it's marked as **False**.
2. **Summing Up Null Values**:
   - **.sum()**: This function is chained after **df.isnull()**. It operates column-wise (by default) on the DataFrame returned by **df.isnull()**. Since **True** is treated as 1 and **False** as 0, summing up these values effectively counts the number of **True** values (i.e., null values) in each column.
3. **Printing the Results**:
   - **print(...)**: This function then prints the resulting sums. The output is a series where each index corresponds to a column name from **df**, and the associated value is the count of null values in that column.

In summary, this line of code provides a count of missing values for each column in the DataFrame **df**, helping in the assessment of data quality and determining if data cleaning or imputation is needed.

```
dict = {}
for i in list(df.columns):
    dict[i] = df[i].value_counts().shape[0]

pd.DataFrame(dict,index=["unique count"]).transpose()
```

Here we are calculating the number of unique values for each column in the DataFrame **df** and then presenting this information in a new DataFrame. Here's a step-by-step explanation:

1. **Creating an Empty Dictionary**:
    - **dict = {}**: Initializes an empty dictionary named **dict**.
2. **Looping Over DataFrame Columns**:
    - **for i in list(df.columns)**: This loop iterates over each column name in the DataFrame **df**. The **df.columns** attribute returns an Index object containing the column labels of the DataFrame.
3. **Counting Unique Values**:
    - **df[i].value_counts().shape[0]**: For each column **i** in **df**, the method **value_counts()** counts the occurrence of each unique value. The resulting object is a Series containing counts of unique values.
    - **.shape[0]**: This part of the code retrieves the length of this Series, which is the count of unique values in the column. Essentially, it's the number of different values that appear in that column.
4. **Storing Results in the Dictionary**:
    - **dict[i] = ...**: The unique value count for each column is stored in the **dict** dictionary with the column name **i** as the key.
5. **Creating a New DataFrame**:
    - **pd.DataFrame(dict,index=["unique count"]).transpose()**: This line creates a new DataFrame from the dictionary **dict**. The keys of the dictionary become the column labels of the DataFrame, and the corresponding values are the unique counts.
    - **index=["unique count"]**: Sets the index of this DataFrame with a single index label **"unique count"**.
    - **.transpose()**: Transposes the DataFrame. This means that the column labels (originally the keys of **dict**, representing each column in **df**) become the index (row labels) of the new DataFrame, and the unique counts are displayed as values in a single row.

The final output is a DataFrame where each row corresponds to a column in the original DataFrame **df**, and there is a single column **"unique count"** that lists the number of unique values in each of those columns. This is useful for understanding the diversity of data in each column of the DataFrame.

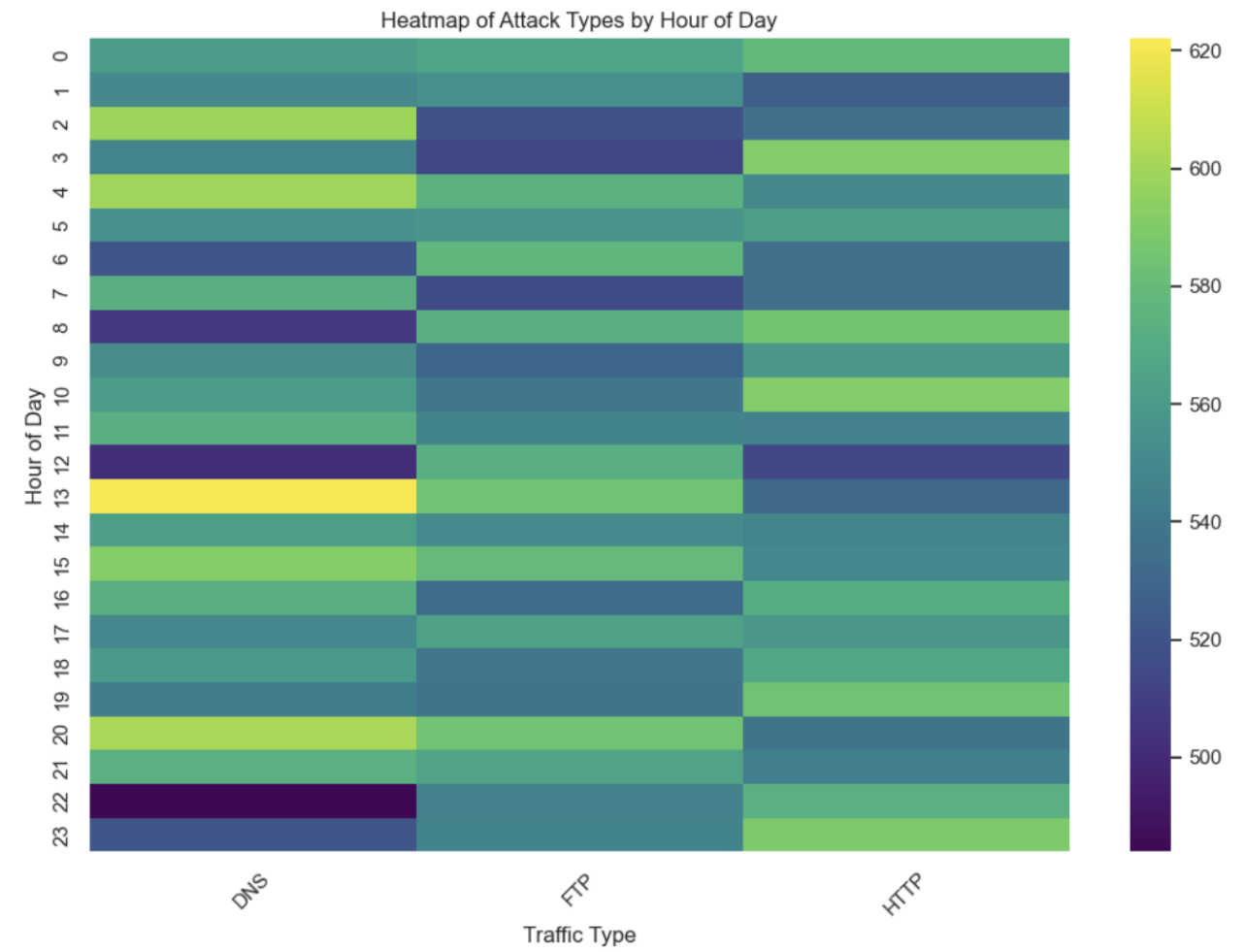
Heatmap:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Load the dataset (replace with the actual file path)
csv_file_path = '/Users/shrutikrishnapagadala/Downloads/cybersecurity_attacks.csv'
df_attacks = pd.read_csv(csv_file_path)

# Convert 'Timestamp' to datetime and extract the hour of the day
df_attacks['Hour'] = pd.to_datetime(df_attacks['Timestamp']).dt.hour

# Create a pivot table to count occurrences of each traffic type for each hour
pivot_table = df_attacks.pivot_table(index='Hour', columns='Traffic Type', aggfunc='size', fill_value=0)

# Plotting the heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(pivot_table, annot=False, cmap='viridis')
plt.title('Heatmap of Attack Types by Hour of Day')
plt.ylabel('Hour of Day')
plt.xlabel('Traffic Type')
plt.xticks(rotation=45)
plt.show()
```

The provided code is a Python script using the Pandas, Seaborn, and Matplotlib libraries to analyze and visualize data from a cybersecurity dataset, focusing on the distribution of different types of network traffic over the hours of the day. Here's a step-by-step breakdown of its functionality:

1. **Import Libraries**:
    - **import pandas as pd**: Imports the Pandas library, which is used for data manipulation and analysis.
    - **import seaborn as sns**: Imports the Seaborn library, a Python visualization library based on Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.
    - **import matplotlib.pyplot as plt**: Imports the Pyplot module from Matplotlib, a plotting library for Python and its numerical mathematics extension NumPy.
2. **Load the Dataset**:
    - **df_attacks = pd.read_csv(csv_file_path)**: Loads a CSV file containing cybersecurity attack data into a Pandas DataFrame named **df_attacks**. The **csv_file_path** variable holds the file path to the dataset.
3. **Data Preprocessing**:
    - **df_attacks['Hour'] = pd.to_datetime(df_attacks['Timestamp']).dt.hour**: This line converts the 'Timestamp' column of the DataFrame into datetime objects and then extracts the hour component. This is stored in a new column called 'Hour', which represents the hour of the day when each event (row) in the dataset occurred.
4. **Create a Pivot Table**:
    - **pivot_table = df_attacks.pivot_table(index='Hour', columns='Traffic Type', aggfunc='size', fill_value=0)**: Creates a pivot table from the DataFrame. This table has hours of the day as its index and different 'Traffic Types' as columns. The values in the table are counts (**size**) of occurrences of each traffic type for each hour. The **fill_value=0** argument replaces NaN values with 0, indicating no occurrences for that particular hour and traffic type.
5. **Plotting the Heatmap**:
    - **plt.figure(figsize=(12, 8))**: Initializes a figure for plotting with a specified size (12 inches wide by 8 inches high).
    - **sns.heatmap(pivot_table, annot=False, cmap='viridis')**: Creates a heatmap using Seaborn. The heatmap represents the data from the pivot table, with the 'viridis' colormap. The **annot=False** argument specifies that the heatmap should not annotate cells with the numeric values they represent.
    - **plt.title(...)**, **plt.ylabel(...)**, **plt.xlabel(...)**: These lines set the title of the heatmap and the labels for the y-axis and x-axis. The title is 'Heatmap of Attack Types by Hour of Day', and the axis labels are 'Hour of Day' and 'Traffic Type'.
    - **plt.show()**: Displays the heatmap.

This heatmap visualization is useful for understanding patterns in cybersecurity attacks or network traffic, such as identifying peak hours for different types of traffic or attacks. It can provide insights into the daily cycle of network activity, which can be valuable for cybersecurity analysis.



The output is a heatmap visualization that represents the frequency of different types of network traffic (or attacks) across various hours of the day. The x-axis is labeled "Traffic Type" and lists types such as DNS, FTP, and HTTP. The y-axis is labeled "Hour of Day" and ranges from 0 to 23, representing the 24 hours in a day.

Each cell in the heatmap corresponds to a particular hour and traffic type, with the color of the cell indicating the frequency of events recorded. The color legend on the right shows a gradient from light colors to dark colors, with a scale that appears to range from 500 to 620, suggesting that these are the minimum and maximum values of the data points in the dataset. Darker colors, particularly purple and dark blue, represent higher frequencies, whereas lighter colors represent lower frequencies.

From the visual output, it appears that there are specific hours where certain types of traffic are more frequent. For instance, there seems to be a noticeable concentration of events for one type of traffic in the late hours (around 22 or 23 on the y-axis), indicated by a yellowish cell, which could suggest a peak in activity for that type of traffic at that time. However, the specific numbers and precise patterns are not visible because annotations (numeric values in the cells) are not enabled.

This heatmap is useful for identifying trends and patterns in network activity, which can be critical for planning cybersecurity measures, such as allocating resources for monitoring and defense during peak hours of malicious activity.

Second heatmap

```
# Additional imports
import calendar

# Convert 'Timestamp' to datetime (if not already converted)
df_attacks['Timestamp'] = pd.to_datetime(df_attacks['Timestamp'])

# Extract day of the week and month from 'Timestamp'
df_attacks['DayOfWeek'] = df_attacks['Timestamp'].dt.day_name()
df_attacks['Month'] = df_attacks['Timestamp'].dt.month.apply(lambda x: calendar.month_abbr[x])

# Creating pivot tables for day of the week and month
pivot_day_of_week = df_attacks.pivot_table(index='DayOfWeek', columns='Traffic Type', aggfunc='size', fill_value=0)
pivot_month = df_attacks.pivot_table(index='Month', columns='Traffic Type', aggfunc='size', fill_value=0)

# Plotting the heatmap for day of the week
plt.figure(figsize=(12, 8))
sns.heatmap(pivot_day_of_week, annot=False, cmap='viridis')
plt.title('Heatmap of Attack Types by Day of the Week')
plt.ylabel('Day of the Week')
plt.xlabel('Traffic Type')
plt.show()

# Plotting the heatmap for month of the year
plt.figure(figsize=(12, 8))
sns.heatmap(pivot_month, annot=False, cmap='viridis')
plt.title('Heatmap of Attack Types by Month of the Year')
plt.ylabel('Month of the Year')
plt.xlabel('Traffic Type')
plt.xticks(rotation=45)
plt.show()
```

The code snippet performs data manipulation and visualization tasks on a cybersecurity dataset with Python using Pandas, Seaborn, Matplotlib, and the calendar module. Here's what each part of the code is doing:

1. **Import the calendar module**: This module provides useful functions related to the calendar, such as retrieving month names.
2. **Convert 'Timestamp' to datetime**:
    - **df_attacks['Timestamp'] = pd.to_datetime(df_attacks['Timestamp'])**: Converts the 'Timestamp' column in the **df_attacks** DataFrame to datetime objects, if they aren't already.
3. **Extract Day of the Week and Month**:
    - **df_attacks['DayOfWeek'] = df_attacks['Timestamp'].dt.day_name()**: Creates a new column 'DayOfWeek' in the DataFrame by extracting the day of the week from the 'Timestamp' column and converting it to the day's name (e.g., Monday, Tuesday, etc.).
    - **df_attacks['Month'] = df_attacks['Timestamp'].dt.month.apply(lambda x: calendar.month_abbr[x])**: Creates a new column 'Month' by extracting the month from the 'Timestamp' column. It then uses the **calendar.month_abbr** list (which contains abbreviated month names) to convert the month number to an abbreviated month name.
4. **Create Pivot Tables**:
    - **pivot_day_of_week = df_attacks.pivot_table(index='DayOfWeek', columns='Traffic Type', aggfunc='size', fill_value=0)**: Creates a pivot table that summarizes the size (i.e., count) of different 'Traffic Types' for each day of the week.

- **pivot_month = df_attacks.pivot_table(index='Month', columns='Traffic Type', aggfunc='size', fill_value=0)**: Similarly, creates another pivot table that summarizes the count of 'Traffic Types' for each month.

5. **Plotting Heatmaps**:
   - Two separate heatmaps are created, one for the pivot table by day of the week and another for the pivot table by month. Both use the 'viridis' color map and do not annotate the cells with their numeric values (**annot=False**).
   - The first heatmap has the title 'Heatmap of Attack Types by Day of the Week' with 'Day of the Week' on the y-axis and 'Traffic Type' on the x-axis.
   - The second heatmap is titled 'Heatmap of Attack Types by Month of the Year', with 'Month of the Year' on the y-axis and 'Traffic Type' on the x-axis. The x-ticks are rotated 45 degrees for better readability.
   - Each heatmap is displayed using **plt.show()**.

These visualizations can provide insights into the patterns of network traffic or attack types based on different timescales, such as daily or monthly trends. This can help in identifying specific days or months that might have higher traffic or attack rates, which is valuable information for cybersecurity monitoring and defense planning.

The first heatmap displays the frequency of different types of network traffic or attacks by day of the week. The days are ordered vertically from Friday to Wednesday, which suggests that they may not be in the usual chronological order we expect (Sunday to Saturday). The traffic types are displayed horizontally with labels such as DNS, FTP, and HTTP. The color intensity represents the frequency of attacks, with a color scale ranging from 1820 to 1960. Days or traffic types with darker colors (closer to purple) have higher frequencies of attacks. For example, there is a particularly high frequency of one type of traffic on Saturday, as indicated by the dark purple color.

The second heatmap shows the frequency of different traffic types by month of the year. The months are listed vertically and seem to be out of the usual January-to-December order. The same traffic types are displayed horizontally. The color scale for this heatmap ranges from 900 to 1250. As with the first heatmap, darker colors indicate higher frequencies of attacks. For instance, there appears to be a peak for one traffic type in February, shown by the dark purple color.

In both heatmaps, the exact values are not visible because the **annot** parameter is set to **False**, which means the individual cell values are not annotated on the heatmap. These visualizations are useful for identifying patterns and trends in attack frequencies related to days of the week and months of the year, which could guide cybersecurity strategies and resource allocation. However, for more detailed analysis, it would be beneficial to have the days and months in their standard order and possibly enable annotations to see exact frequencies.

Barplot:-

This code defines a function named **bar_plot** for visualizing the frequency distribution of a categorical variable from a DataFrame using a bar chart. Here's a breakdown of its functionality:

1. **Function Definition**:
   - **def bar_plot(variable)**: This line defines a function **bar_plot** that takes one argument, **variable**, which is expected to be the name of a column in the DataFrame **df**.
2. **Extracting Data for the Specified Variable**:
   - **var = df[variable]**: This line extracts the data from the column named **variable** in the DataFrame **df**.
3. **Counting Category Frequencies**:
   - **varValue = var.value_counts()**: Computes the frequency of each unique value in the column **var**. The result **varValue** is a pandas Series where the index represents unique values and the corresponding values are their frequencies in the dataset.
4. **Setting Up the Visualization**:
   - **plt.figure(figsize=(10, 5))**: Creates a new figure for the plot with the specified size (10 inches wide, 5 inches tall).
5. **Creating the Bar Chart**:
   - **plt.bar(varValue.index, varValue, color='green')**: Plots a bar chart using **varValue**. The x-axis uses the indices of **varValue** (the unique values of the variable), the height of the bars corresponds to their frequencies, and the bars are colored green.
   - The comment about the colormap suggests an alternative way to color the bars using a colormap (**plt.cm.Paired**), but this is not used in the current implementation.
6. **Customizing the Plot**:
   - **plt.xticks(varValue.index, varValue.index.values)**: Sets the tick labels on the x-axis to match the unique values of the variable.
   - **plt.ylabel("Frequency")**: Sets the label for the y-axis.
   - **plt.title(variable)**: Sets the title of the plot to be the name of the variable.
7. **Displaying the Plot and Output**:
   - **plt.show()**: Displays the bar chart.
   - **print("{}: \n {}".format(variable, varValue))**: Prints the name of the variable and its frequency distribution as text output.

Overall, this function is a utility for exploring and visualizing the distribution of categorical data in a DataFrame. It helps in understanding how often each unique value appears in the specified column.
This code snippet is iterating through a list of column names in a DataFrame **df** and printing the value counts for each of these columns. Here's a breakdown of its functionality:

1. **Importing pandas**:
   - **import pandas as pd**: This line imports the pandas library, a powerful tool for data manipulation and analysis in Python.
2. **Specifying Column Names**:
   - **category1 = ["Protocol", "Packet Type", "Traffic Type", "Attack Type", "Action Taken", "Severity Level", "Network Segment", "Log Source"]**: This line creates a list named **category1** containing specific column names from the DataFrame **df**.
3. **Iterating Through the Columns**:
   - **for c in category1:**: This for loop iterates over each element in the **category1** list. Each element represents a column name in the DataFrame **df**.

4. **Printing Value Counts**:
    - **print(f"Value Counts for {c}:\n{df[c].value_counts()}\n")**: For each column name **c** in the list, this line prints a formatted string that includes:
        - The column name (**c**).
        - The value counts of each unique entry in that column, obtained by **df[c].value_counts()**. The **value_counts()** function returns a Series containing counts of unique values in the specified column, in descending order so that the most frequently-occurring element is at the top.
        - A newline character **\n** for better readability, separating the output for different columns.

In summary, this code is used for data exploration, specifically to understand the distribution of values in various categorical columns of the DataFrame **df**. It helps in identifying the frequency of each unique value in the specified columns, which can be important for understanding the characteristics of the dataset, especially in fields like data analysis, data science, or machine learning.

```
category = ["Protocol", "Packet Type","Traffic Type","Attack Type", "Action Taken", "Severity Level", "Network Segment", "Log Source"]
for c in category:
    bar_plot(c)
```

1. **Protocol**: Shows the frequency of each network protocol in the dataset. This plot helps in understanding which protocols are most commonly used or targeted in the dataset's cybersecurity incidents.
2. **Packet Type**: Illustrates the distribution of different packet types (like Data, Control). It can reveal the most common packet types involved in the cybersecurity incidents.
3. **Traffic Type**: Displays the count of different traffic types (e.g., HTTP, DNS). It's useful for identifying which types of network traffic are most prevalent in the dataset.
4. **Attack Type**: Provides an overview of the frequency of different attack types. This is crucial for understanding the landscape of cyber threats represented in the dataset.
5. **Action Taken**: Shows what actions were taken in response to the incidents (e.g., Blocked, Logged, Ignored). This plot can indicate the most common responses to cybersecurity incidents.
6. **Severity Level**: Depicts the distribution of severity levels (e.g., Low, Medium, High) of the incidents. This helps in assessing the impact level of the incidents in the dataset.
7. **Network Segment**: Reveals the count of incidents in different network segments. It can highlight which network segments are more frequently involved in cybersecurity incidents.
8. **Log Source**: Shows the distribution of different log sources (e.g., Server, Firewall). It's useful for understanding where most of the logging data is coming from.

Each plot provides insights into the respective aspect of cybersecurity incidents, such as prevalence, types, responses, and impact.

Seaborn plots

```
import seaborn as sns
import matplotlib.pyplot as plt

# Define your custom color palette
custom_palette = ['#FFFFE0', '#008B8B', '#C32148']  # light yellow, dark cyan, light maroon

for col in ['Protocol', 'Packet Type', 'Traffic Type', 'Attack Signature', 'Action Taken', 'Severity Level', 'Network Segment']:
    plt.figure(figsize=(10, 5))
    # Use the custom palette
    sns.countplot(data=df, x=col, hue='Attack Type', palette=custom_palette)
    plt.xticks(rotation=90)  # Rotate the x labels if needed
    plt.show()
```

1. **Import Libraries**:
   - **import seaborn as sns**: Imports the Seaborn library, which is used for making statistical graphics in Python. It's based on Matplotlib and integrates well with pandas data structures.
   - **import matplotlib.pyplot as plt**: Imports the Pyplot module from Matplotlib, a plotting library that provides a MATLAB-like interface.
2. **Define Custom Color Palette**:
   - **custom_palette = ['#FFFFE0', '#008B8B', '#C32148']**: Defines a custom color palette as a list of HEX color codes. These colors are light yellow, dark cyan, and light maroon, respectively.
3. **Looping through Columns for Plotting**:
   - **for col in [...]:**: This loop iterates over a list of column names (like 'Protocol', 'Packet Type', etc.). For each column in this list, the following steps are executed.
4. **Creating a Plot for Each Column**:
   - **plt.figure(figsize=(10, 5))**: Sets up a new figure with a specified size (10 inches wide by 5 inches high).
   - **sns.countplot(data=df, x=col, hue='Attack Type', palette=custom_palette)**: Creates a count plot using Seaborn. This plot will show the count of occurrences for each unique value in the column **col**. The **hue='Attack Type'** parameter means that within each unique value of **col**, counts will be further broken down by the 'Attack Type' column. The **palette=custom_palette** uses the defined custom color palette for the plot.
   - **plt.xticks(rotation=90)**: Rotates the x-axis labels by 90 degrees to make them more readable, especially useful when there are many categories or the labels are long.
   - **plt.show()**: Displays the plot. This command is necessary for each iteration of the loop to show the plot corresponding to the current **col**.

Output:-

The count plots for the selected columns have been generated. Let's analyze them:

1. **Protocol**: This plot shows the distribution of different protocols used in the cyber attacks. Different colors represent different attack types, allowing us to see which protocols are most frequently targeted by each attack type.
2. **Packet Type**: The distribution of packet types (e.g., Data, Control) in relation to different attack types. This can reveal if certain packet types are more susceptible to specific attacks.
3. **Traffic Type**: Shows the types of network traffic (e.g., HTTP, DNS) involved in the attacks. This can help identify which traffic types are more often targeted.

4. **Attack Signature**: Represents the different signatures detected in the attacks, categorized by attack type. It's useful for understanding the variety of attack signatures and their frequencies.
5. **Action Taken**: Illustrates the actions taken (e.g., Blocked, Logged, Ignored) in response to different attack types. This plot can indicate the effectiveness or commonality of certain responses.
6. **Severity Level**: Shows the distribution of the severity level of the incidents, broken down by attack type. This is crucial for understanding the impact level of different attack types.
7. **Network Segment**: This plot displays the distribution of incidents across different network segments, again categorized by attack type. It can highlight if certain segments are more prone to specific types of attacks.

Each of these plots provides a visual representation of the dataset's attributes in relation to the types of cyber attacks. They can be used to draw insights into patterns, frequencies, and relationships within the data, which is vital for cybersecurity analysis and decision-making.

```python
# Let's first read the contents of the uploaded CSV file to understand its structure and data.
import pandas as pd

# File path
file_path = '/Users/shrutikrishnapagadala/Downloads/cybersecurity_attacks.csv'

# Reading the CSV file
cybersecurity_attacks_df = pd.read_csv(file_path)

# Displaying the first few rows of the dataframe to understand its structure
cybersecurity_attacks_df.head()
```

	Timestamp	Source IP Address	Destination IP Address	Source Port	Destination Port	Protocol	Packet Length	Packet Type	Traffic Type	Payload Data	...	Action Taken	Severity Level	User Information	Device Information
0	2023-05-30 06:33:58	103.216.15.12	84.9.164.252	31225	17616	ICMP	503	Data	HTTP	Qui natus odio asperiores nam. Optio nobis ius...	...	Logged	Low	Reyansh Dugal	Mozilla/5.0 (compatible; MSIE 8.0; Windows NT ...
1	2020-08-26 07:08:30	78.199.217.198	66.191.137.154	17245	48166	ICMP	1174	Data	HTTP	Aperiam quos modi officiis veritatis rem. Omni...	...	Blocked	Low	Sumer Rana	Mozilla/5.0 (compatible; MSIE 8.0; Windows NT ...
2	2022-11-13 08:23:25	63.79.210.48	198.219.82.17	16811	53600	UDP	306	Control	HTTP	Perferendis sapiente vitae soluta. Hic delectu...	...	Ignored	Low	Himmat Karpe	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT ...

The code is performing the following operations using the Pandas library, a powerful data manipulation tool in Python:

1. **Import Pandas Library**:
   - **import pandas as pd**: This line imports the Pandas library and gives it the alias 'pd', which is a conventional alias for Pandas. This allows you to use Pandas functions by referencing them with **pd**.
2. **Set the File Path**:
   - **file_path = '/path/to/cybersecurity_attacks.csv'**: This line stores the file path of the CSV file you want to analyze in a variable named **file_path**.
3. **Read the CSV File**:

- **cybersecurity_attacks_df = pd.read_csv(file_path)**: This line reads the CSV file located at the specified **file_path** into a Pandas DataFrame. A DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

4. **Display the DataFrame**:
   - **cybersecurity_attacks_df.head()**: This line displays the first five rows of the DataFrame by default. The **.head()** method is useful for getting a quick overview of the dataset, including the structure of the DataFrame, the column names, and the type of data contained in the first few rows.

The code does not actually change any data or perform any analysis. Its purpose is to load and display the initial part of the dataset so that you can get a sense of what the data looks like and decide on the next steps for analysis or manipulation.

Decision Tree:-

The provided code snippet is implementing a machine learning workflow using a Decision Tree classifier within the scikit-learn library in Python. Here's a breakdown of what each part of the code is doing:

1. **Import DecisionTreeClassifier**:
   - The **DecisionTreeClassifier** class is imported from scikit-learn's **tree** module. Decision Trees are a type of supervised learning algorithm that is mainly used for classification tasks.
2. **Create a Pipeline with Decision Tree Classifier**:
   - A pipeline named **model_dt** is created using scikit-learn's **Pipeline** utility. This pipeline includes two main steps:
     - **'preprocessor'**: This refers to a preprocessing step that you have previously defined (but not shown in the snippet). This step typically includes data transformations like scaling, normalization, or encoding.
     - **'classifier'**: The classifier step is set to a **DecisionTreeClassifier** with a **random_state** of 42. Setting **random_state** ensures that the model's splits are reproducible.
3. **Fit the Decision Tree Model**:
   - The pipeline is trained using the **fit** method on the training data (**X_train** and **y_train**). This step involves the decision tree learning from the training data, figuring out how to make decisions and splits based on the features and their relation to the target variable.
4. **Make Predictions with the Decision Tree Model**:
   - The **predict** method is used to make predictions on the test dataset (**X_test**). The model uses the knowledge it gained during training to predict the target variable for each instance in the test set.
5. **Generate and Display a Classification Report**:
   - **classification_report** generates a report on the performance of the classifier. It compares the true labels (**y_test**) with the predicted labels (**y_pred_dt**) and provides a detailed performance analysis that includes metrics like precision, recall, and f1-score for each class.
   - **label_encoder.classes_** is used to provide the actual class names for the report, assuming **label_encoder** is a previously defined and fitted **LabelEncoder** instance.

- The output of **classification_report**, stored in **report_dt**, is likely printed or displayed to review the model's performance.

Overall, the code sets up, trains, and evaluates a Decision Tree classifier on a given dataset, aiming to predict a target variable based on a set of features. The classification report at the end helps in understanding how well the model performs across different classes.

RandomForest:-

The code is modifying a machine learning workflow to focus on a subset of features deemed most relevant for predicting the severity level of cybersecurity attacks. This is a common practice known as feature selection, where the goal is to improve model performance or reduce overfitting by using only the most informative features. Here's what the code is doing step by step:

1. **Feature Selection**:
   - A list of **selected_features** is defined, which includes 'Protocol', 'Packet Length', 'Traffic Type', 'Action Taken', and 'Network Segment'. These are the features that have been determined to be most likely to influence the target variable, 'Severity Level'.
2. **Update the Feature Set**:
   - **X_selected** is created as a subset of the DataFrame **cybersecurity_attacks_df**, containing only the columns specified in **selected_features**.
3. **Splitting the Data**:
   - The dataset is split into training and testing sets using the selected features (**X_selected**) and the previously encoded target variable (**y_encoded**).
   - The **train_test_split** function reserves 30% of the data for testing (**test_size=0.3**) and uses a **random_state** of 42 for reproducibility.
4. **Updating the Preprocessor**:
   - The preprocessor is updated to only consider the selected numerical and categorical features (**numerical_features_sel** and **categorical_features_sel**).
   - The **ColumnTransformer** is adjusted accordingly to apply the **numerical_transformer** and **categorical_transformer** pipelines only to the selected features.
5. **Creating a New Pipeline**:
   - A new pipeline **model_sel** is created with the updated preprocessor **preprocessor_sel** and a **RandomForestClassifier**.
   - The random forest classifier is initialized with the same **random_state** to ensure the results are consistent.
6. **Fitting the Model**:
   - The new pipeline is fitted to the training data, which now only includes the selected features (**X_train_sel** and **y_train_sel**).
7. **Predicting and Evaluating the Model**:
   - The fitted model is used to make predictions on the test data (**X_test_sel**).
   - A classification report is generated using the true labels (**y_test_sel**) and the predicted labels (**y_pred_sel**).
   - The report includes metrics such as precision, recall, and f1-score for each class and uses the original class names provided by **label_encoder.classes_**.
8. **Display the Report**:

- **report_sel** contains the text of the classification report, which is the output of the code and will show the performance of the model with the selected features on the test set.

By focusing on a subset of features, the code aims to refine the model's predictive accuracy and potentially improve its generalization to new, unseen data. Feature selection can often lead to better model interpretability and faster training times.

KNN:-

The code provided is adapting the previous machine learning workflow to utilize a K-Nearest Neighbors (KNN) classifier instead of the Random Forest classifier. This change is part of the experimentation process in machine learning where different models are tried to find the one that performs best for the task at hand. Here's what each part of the code is doing:

1. **Import KNeighborsClassifier**:
   - The **KNeighborsClassifier** class is imported from scikit-learn, which implements the k-nearest neighbors vote classification.
2. **Create a New Pipeline for KNN**:
   - A pipeline named **model_knn** is created with two steps: 'preprocessor' and 'classifier'.
   - The 'preprocessor' uses the previously defined **preprocessor_sel** that includes the transformation steps for the selected features.
   - The 'classifier' is now set to **KNeighborsClassifier()** which initializes a KNN classifier with default parameters.
3. **Fit the KNN Model**:
   - The **model_knn** pipeline is then fitted to the training data (**X_train_sel** for the selected features and **y_train_sel** for the encoded target variable).
4. **Predict and Evaluate the KNN Model**:
   - After fitting, the KNN model is used to predict the target variable for the test set (**X_test_sel**).
   - A classification report (**report_knn**) is generated using **classification_report** from scikit-learn. It compares the predicted labels (**y_pred_knn**) with the true labels (**y_test_sel**) and provides a text report showing the main classification metrics (precision, recall, f1-score, and support) for each class.
   - The **target_names** parameter in the classification report is set to the class names decoded by **label_encoder.classes_**, which makes the report more interpretable by showing the actual class names instead of encoded integers.

The classification report is stored in **report_knn** and will typically be printed or logged to review the model's performance. The KNN classifier is a simple, instance-based learning algorithm that predicts the class of a given data point by looking at the 'k' closest labeled data points and taking a majority vote on their labels. It's often a strong baseline model for classification tasks.

Comparing all three models :-

The code is evaluating and comparing the performance of three different machine learning classification models by calculating their accuracy on a test set. Here's the process:

1. **Import accuracy_score**:

- The function **accuracy_score** from scikit-learn's module **metrics** is imported. This function is used to compute the accuracy of a model, which is the fraction of correct predictions out of all predictions made.

2. **Calculate Accuracies**:
   - **accuracy_rf_sel**: Calculates the accuracy of the Random Forest classifier that was trained on the selected features (**y_pred_sel** vs **y_test_sel**).
   - **accuracy_knn**: Calculates the accuracy of the K-Nearest Neighbors classifier (**y_pred_knn** vs **y_test_sel**).
   - **accuracy_dt**: Assumes there is a Decision Tree model's predictions available as **y_pred** (though the code for its prediction is not shown in the snippet provided) and calculates its accuracy (**y_pred** vs **y_test_sel**).
3. **Create a Dictionary for Comparison**:
   - **model_accuracies_comparison**: A dictionary is created with model names as keys and their respective accuracies as values.
4. **Display the Accuracies**:
   - The dictionary **model_accuracies_comparison** is presumably returned or printed at the end of the code snippet to display the accuracies of the KNN, Decision Tree, and Random Forest models side by side.

By comparing these accuracies, one can determine which model performs best on the test data based on the accuracy metric. Accuracy is a simple and intuitive performance measure, but it should be noted that it may not always be the best metric for evaluating performance, especially in cases where the class distribution is imbalanced. In such cases, other metrics like precision, recall, f1-score, and confusion matrix are also considered for a comprehensive evaluation.

Output of the classifiers:-

The provided accuracy scores for the three machine learning models—K-Nearest Neighbors (KNN), Decision Tree, and Random Forest—indicate their performance in a classification task. Here's an analysis of what these scores tell us:

1. **Overall Low Accuracy**:
   - All models show an accuracy of around 32% to 33%. This level of accuracy is relatively low, especially for binary or multi-class classification tasks where we typically expect higher accuracy for a well-performing model. An accuracy closer to or below 50% suggests that the models are not much better (or are only marginally better) than a random guess in a binary classification scenario.
2. **Comparative Performance**:
   - The Decision Tree model has the highest accuracy at 33.3%, but this is only marginally better than the KNN and Random Forest models, which have accuracies of 32.4% and 32.8%, respectively. The closeness of these accuracy scores suggests that none of the models is significantly outperforming the others.
3. **Implications for Model Selection and Data**:
   - The similar performance of all three models might indicate that the challenge lies within the data itself, such as lack of informative features, high noise, or data that doesn't differentiate well between classes.

- It's also possible that the models are underfitting, meaning they are too simple to capture the complexity and patterns in the data. This is especially a concern with the Decision Tree, which is prone to high variance and might not generalize well.
- Another possibility is that the dataset is imbalanced, which can skew accuracy. If one class significantly outnumbers another, a model might appear to have a decent accuracy by merely predicting the majority class most of the time.

4. **Considerations for Improvement**:
   - **Feature Engineering**: Investigate if the features used are adequate and informative enough. More relevant features might need to be identified.
   - **Model Complexity**: Experiment with adjusting the complexity of the models. For instance, deeper trees or different parameters in the Random Forest might yield better results.
   - **Alternative Metrics**: In cases of imbalanced data, accuracy might not be the best performance metric. Metrics like precision, recall, F1-score, ROC-AUC might provide a better insight into the model's performance.
   - **Cross-validation**: Ensure the model's performance is validated using techniques like cross-validation rather than a single train-test split to get a more reliable estimate of its performance.

In summary, the accuracy scores suggest that further investigation and refinement of the models and data are required for better performance. Adjustments in the modeling approach, deeper data analysis, and exploring different evaluation metrics are recommended steps forward.