Subject: Foundations of Data Science

Faculty: Prof. Saravanakumar K

Project on –

Malware Detection in an Android Application using Permission Requirements

Submitted by –

Uddipan Sarkar 21BCE0755

Aman Sood 21BCE3623

Kshitij Radotra 21BCE2695

Mitta Yashwanth Sai 21BCE0230

Srija Pal 21BCE2307

Himanshu Keshav Parab 21BDS0263

Satvik Sachin 21BCE0423

CONTENTS

1. Overview of the Project

1.1 Introduction:

This project focuses on the detection of malware in Android applications by analyzing their permission requirements. With the increasing popularity of mobile devices, particularly Android smartphones, the risk of malware attacks has also grown. Malicious apps often request excessive permissions, which can be indicative of their malicious intent. This project aims to develop a system that automatically identifies potentially harmful applications based on their permission requirements.

1.2 Data Discovery:

The initial phase of the project involved collecting a diverse dataset of Android applications from various sources, including app repositories and marketplaces. The dataset includes information about each application's permissions, features, and labels indicating whether it is benign or malicious. Data discovery enabled us to understand the characteristics of Android applications and identify relevant features for malware detection.

1.3 Data Collection and Preparation:

The collected dataset underwent preprocessing to ensure its quality and suitability for analysis. This phase involved cleaning the data, handling missing values, and transforming the features into a format suitable for modeling. Additionally, we performed feature engineering to extract relevant information from the permission requirements of each application, which served as the basis for malware detection.

1.4 Model Planning:

In the model planning phase, we defined the objectives of the analysis and selected appropriate techniques for malware detection. We formulated hypotheses about the relationship between permission requirements and the presence of malware in Android applications. Based on these hypotheses, we designed a machine learning approach to classify applications as benign or malicious using their permission patterns.

1.5 Model Building:

Using the prepared dataset, we trained and evaluated machine learning models to predict the presence of malware in Android applications. We experimented with various algorithms, including decision trees, random forests, and support vector machines, to identify the most effective approach for malware detection. Model building involved iterative experimentation and optimization to improve the accuracy and performance of the classifiers.

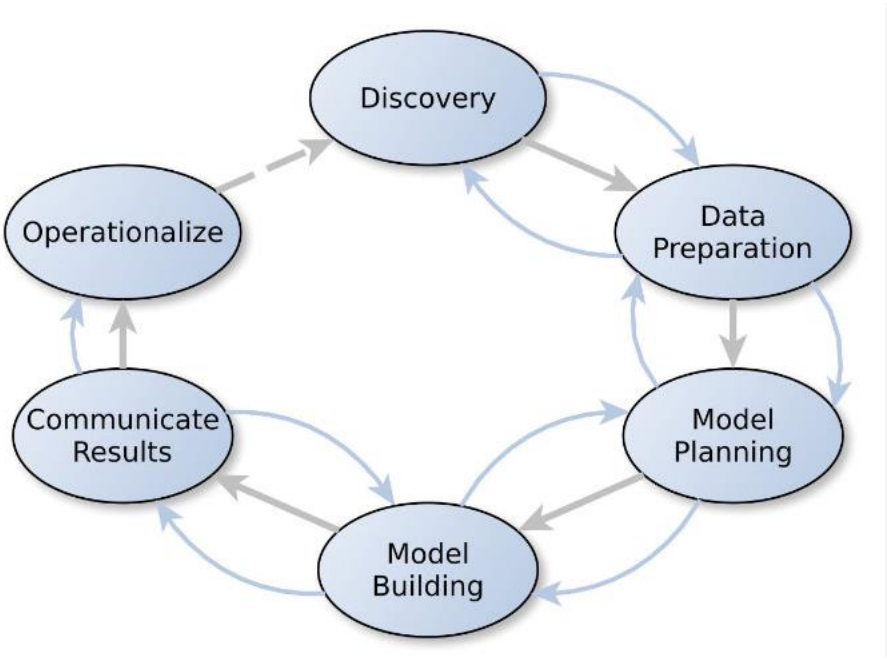
1.6 Communicate Results:

After developing the malware detection models, we communicated the results and insights derived from the analysis. We summarized the performance of the models, including metrics such as accuracy, precision, recall, and F1-score. Additionally, we interpreted the key features and patterns identified by the models to understand the characteristics of malicious Android applications better.

1.7 Operationalize:

The final phase of the project involved operationalizing the malware detection system for practical use. We integrated the developed models into a software application or service that can analyze Android applications in real-time and identify potential threats based on their permission requirements. Operationalizing the system enables proactive detection and mitigation of malware in Android ecosystems, enhancing the security of mobile devices.

1.8 Diagrammatic representation:



2. Phase 1: Data Discovery

2.1 Introduction:

Phase 1 of the project focuses on data discovery, where the initial hypotheses are formulated, potential data sources are identified, the business domain is explored, the problem is framed, and key stakeholders are identified. This phase sets the foundation for subsequent data collection and analysis activities.

2.2 Initial Hypothesis:

- Applications with excessive and unnecessary permission requirements are more likely to contain malware.
- Malware authors often exploit permissions granted to apps for malicious activities such as stealing personal information or accessing sensitive data.

2.3 Identifying Potential Data Sources:

1. Google Play Store
2. Third-party app markets
3. Malware analysis platforms
4. App metadata checks for insights on malware signatures and behaviors.

2.4 Learning the Business Domain:

- Understanding app permissions and familiarizing with malware detection techniques.
- Checking regulatory frameworks to ensure compliance.
- Understanding different types of permissions and existing methodologies for detecting malware.

2.5 Resources Available:

- Application metadata, developers, and cybersecurity experts.
- Obtaining permission data from sources.
- Understanding tactics employed by malicious users to exploit app permissions.

2.6 Framing the Problem:

- **Goal:** Automatically detect potentially malicious apps based on their permission requirements.
- **Approach:** Analyzing permission patterns to identify common characteristics associated with malware.
- **Criteria:** Developing a basis to quantify the risk associated with different permissions.

- **Result:** Model predicting the likelihood of an app being malware based on its permission requirements.

2.7 Identifying Key Stakeholders:

1. Application Users: Their feedback and preferences are crucial.
2. Application Developers: Understanding app functionality aids in distinguishing genuine features from potential malware behavior.
3. Regulatory Bodies: Enforcing rules related to app permissions and data privacy.
4. Cybersecurity Personnel: Help in vulnerability analysis and inform strategies for mitigating emerging risks.

3. Phase 2: Data Collection and Preparation

3.1 Introduction:

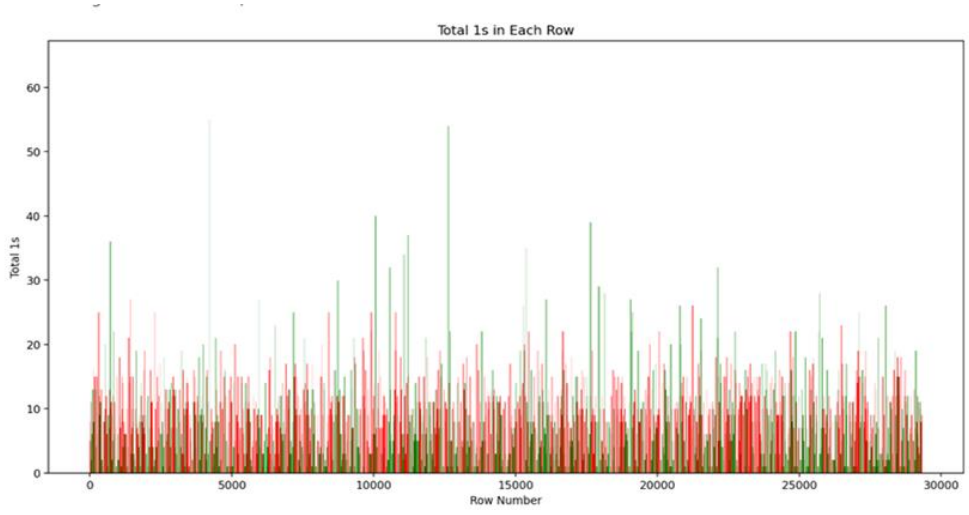
Phase 2 of the project involved the collection and preparation of data for further analysis. This phase aimed to establish a clean and structured dataset suitable for training machine learning models to detect malware in Android applications based on their permission requirements.

3.2 Preparing the Analytic Sandbox:

An isolated environment was set up in VSCode to facilitate data exploration and analysis without affecting production systems. This allowed analysts to work with the data in a controlled environment.

3.3 Performing ELT (Extract, Load, Transform) and Outlier Visualization:

The initial dataset comprised 29,335 unique data entries with 86 feature columns representing permissions and 1 result column indicating predictions. Outlier visualization techniques were employed to identify abnormal values, distinguishing malware from non-malware entries.



3.4 Learning about the Data:

An initial hypothesis was formulated, suggesting that applications with excessive permissions are more likely to contain malware. The dataset contained information about permissions used by applications and their corresponding malware presence.

3.5 Data Conditioning:

Data quality checks revealed no missing values, and the dataset was in binary format. Outlier removal techniques were applied to eliminate entries with unusually high permission counts but were labelled as non-malware. Standardization using StandardScaler ensured that each feature had a mean of 0 and standard deviation of 1. The dataset was split into training and testing sets, with 80% used for training and 20% for testing model accuracy.

3.6 Tools Used:

Python was the primary programming language used for data preparation. Packages such as StandardScaler and train_test_split from sklearn were utilized for data normalization and dataset splitting, respectively.

4. Phase 3: Model Planning

4.1 Introduction:

Phase 3 of the project involves planning the models for detecting malware presence in Android applications based on their permission requirements. This phase focuses on feature selection and the selection of appropriate model types for binary classification.

4.2 Feature Selection:

- **Initial features:** The dataset initially comprised 86 features.
- **Method:** Z-score analysis was employed to identify features significantly deviating from the mean.
- **Purpose:** The goal was to identify features with high z-scores, indicating significant deviations, which could potentially impact the model's performance.
- **Action:** Two features with the highest z-scores were removed from the dataset.

4.3 Model Types:

Two model types were considered for malware detection:

1. **Regression**
2. **Neural Network**

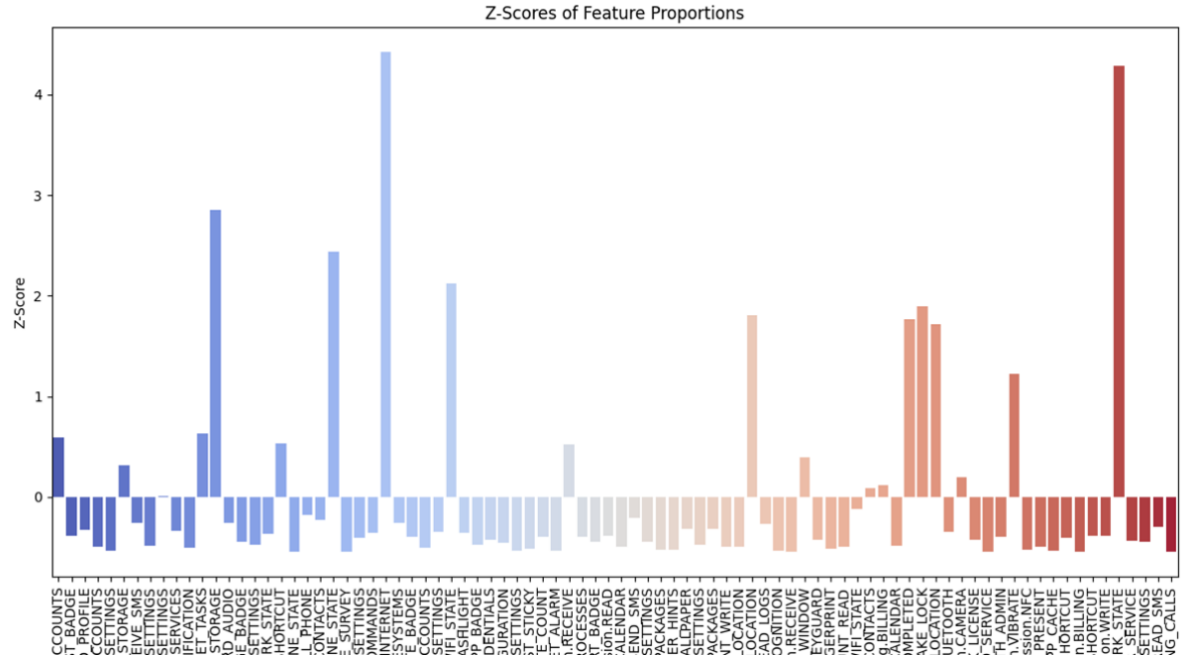
4.4 Reason for Selection:

- **Problem:** The task involves predicting malware presence, a binary classification problem.

- **Data:** The dataset consists of 84 non-continuous binary features.
- **Objective:** The objective is binary classification, determining whether an application is malware or not.
- **Neural Network:** Chosen due to its suitability for binary classification tasks with non-continuous data.

4.5 Visualization for Feature Selection:

- A Z-score plot was utilized for feature analysis.
- Interpretation: High z-scores indicate significant deviations from the mean, suggesting potential feature removal.



4.6 Why SNN over other models:

After careful consideration, a Sequential Neural Network was chosen as the model for malware detection. This decision was based on its suitability for binary classification tasks with non-continuous data, which aligns with the characteristics of the dataset. Regression models were deemed less suitable as they are more appropriate for continuous datasets. Additionally, Convolutional Neural Networks (CNNs) were not chosen due to the binary nature of the dataset, which is not well-suited for CNNs designed for multidimensional data like images or videos.

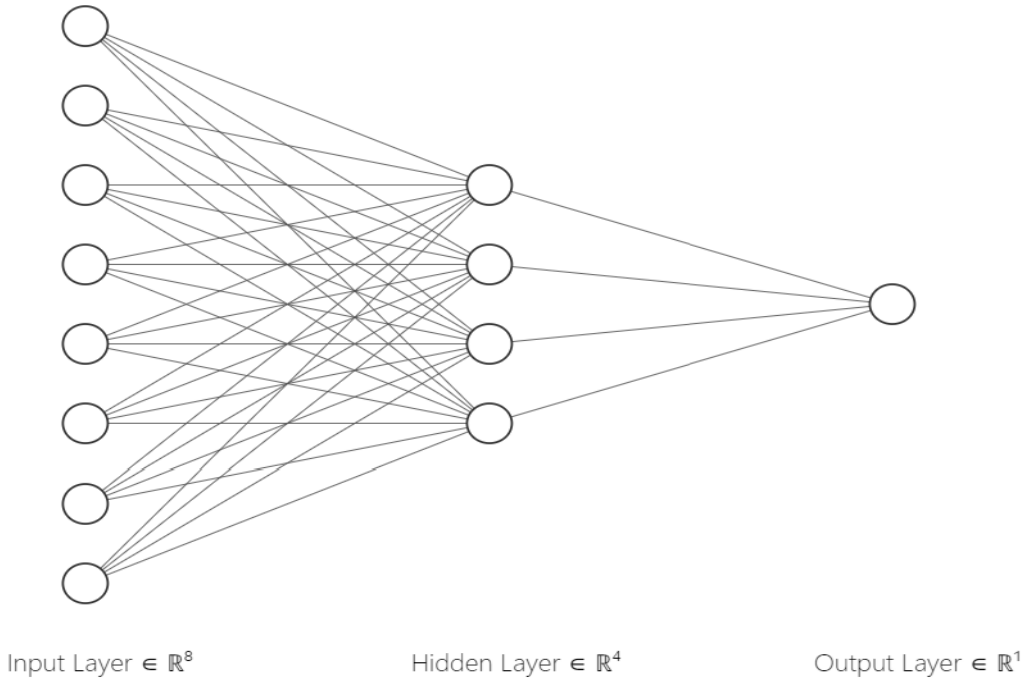
5. Phase 4: Model Building

5.1 Overall Structure:

The model constructed is a sequential neural network with the following architecture:

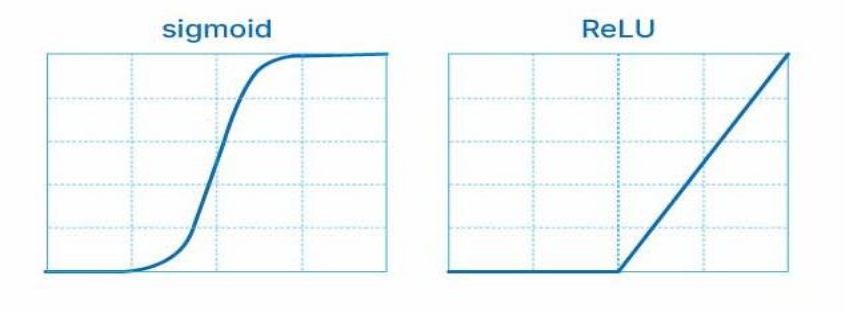
- Input layer: 84 neurons, with each neuron representing a feature.
- Hidden layer: 42 neurons, which is half the number of neurons in the previous layer, aiming to identify tighter relationships between features and the result and prevent overfitting.
- Final layer: 1 neuron, providing a final binary prediction.

5.2 Visual Representation (8:4:1 to fit):



5.3 Other Metrics Used:

- **Activation function for neurons:** Sigmoid and Relu.
- **Loss function:** Binary Cross Entropy.



5.4 Implementation:

- **Language:** Python
- **Framework:** TensorFlow
- **Libraries:** Scikit-learn, Pandas
- **Graphing:** Seaborn, Matplotlib
- **Tools:** VSCode, Excel

5.5 Training and Testing Results:

- Trained for 10 epochs.
- Average Binary Cross Entropy Loss: 0.1008.
- Tested using split data.
- Average accuracy: 0.9680.
- Therefore, with the current structure and metrics, the model achieves an accuracy of 96.8%.

This phase successfully developed a sequential neural network model for detecting malware presence based on permission requirements. The model's architecture, along with the activation and loss functions used, were carefully selected to optimize performance. Training and testing results indicate a high level of accuracy, demonstrating the effectiveness of the model in identifying potentially malicious applications. Further evaluation and refinement may be conducted in subsequent phases to enhance the model's performance and robustness.

6. Phase 5: Communicate Results

6.1 Accuracy:

The accuracy measure represents the overall correctness of model predictions. It indicates the percentage of correct classifications out of the total instances evaluated.

Accuracy of the model is 96.8%

6.2 Precision:

Precision measures the accuracy of the positive predictions of model. It measures the proportion of true positive predictions among all positive predictions made by the model. A high precision score indicates that the model has a low false positive rate, meaning that when it predicts an app as malware, it is correct most of the time.

Precision of the model is 97.43%

6.3 Recall:

Recall assesses the ability of the model to correctly identify all instances of malware. It measures the proportion of true positive predictions among all actual positive instances in the dataset. A high recall score indicates that the model captures a large portion of actual positive instances (malware apps) present in the dataset.

Sensitivity of the model is 96.39%

6.4 F1 Score:

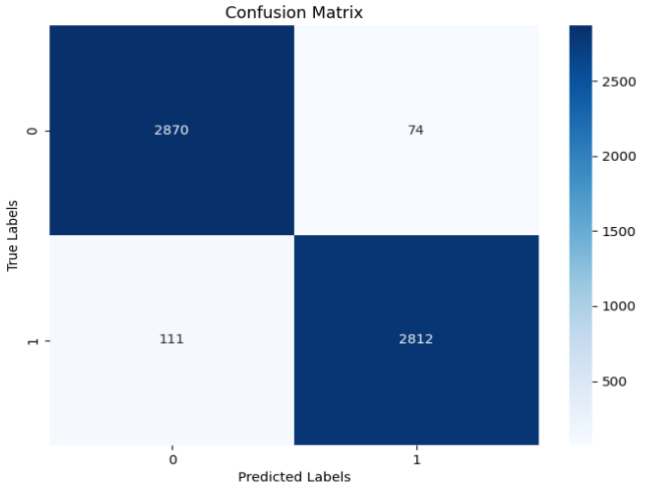
The F1 score is the harmonic mean of precision and recall, providing a balanced measure of a model's performance. A high F1 score indicates both high precision and high recall, striking a balance between minimizing false positives and false negatives.

F1 score of the model is 96.77%

6.5 Confusion Matrix:

The confusion matrix allows a detailed analysis of the models performance shoeint where it excels and areas of improvement. It compares the predicted class labels with the actual class labels for a set of instances.

- True Positives (TP): Instances correctly predicted as positive.
- True Negatives (TN): Instances correctly predicted as negative.
- False Positives (FP): Instances incorrectly predicted as positive.
- False Negatives (FN): Instances incorrectly predicted as negative.



6.6 Applications:

- Mobile Security Solutions: Integration into security apps to provide real-time protection against malware threats.
- App Marketplaces Enhancement: Improving app vetting processes in marketplaces like Google Play to ensure safer app downloads.
- Enterprise Security: Safeguarding corporate devices against malware by integrating into mobile device management solutions.
- Cybersecurity Research: Providing a dataset and model for analyzing malware detection techniques and advancing cybersecurity research.
- Educational Resource: Serving as a learning tool for cybersecurity students and professionals interested in malware detection.

6.7 Limitations:

Identifying limitations in a data analytics project is crucial for providing a comprehensive understanding of the findings and ensuring the accuracy and reliability of the results. Some limitations that apply to the project are:

- Limited availability of high-quality data may restrict the performance and generalizability of the model.
- The dataset size might not adequately represent the diversity of real-world Android apps, leading to biases or overfitting.
- The selection and extraction of features may not capture all relevant information, potentially missing important patterns or relationships in the data.
- Certain features may be redundant, noisy, or irrelevant, impacting the model's performance.
- Evaluation metrics such as precision, recall, and F1 score may not adequately capture the trade-offs between false positives and false negatives.
- Changes in the Android ecosystem, such as updates to permissions or new malware variants, could impact the model's effectiveness over time.
- External factors beyond the scope of the project, such as changes in user behavior or technological advancements, may influence the relevance and applicability of the findings.
- Assumptions made during the project, such as the independence of features or the stationarity of the data distribution, may not hold true in practice.

6.8 Recommendations:

- Validate the model's performance on additional datasets to ensure its robustness.
- Assess the feasibility of deploying the model in real-world scenarios, considering factors such as computational resources and ethical implications.

- Continuously monitor and update the model to maintain its effectiveness over time.
- Explore additional sources of data to supplement the existing dataset, such as incorporating information about app ratings, reviews, or developer reputation.
- Experiment with different feature engineering techniques, such as creating new features based on combinations of existing ones or incorporating domain-specific knowledge, to enhance the model's performance.
- Conduct feature importance analysis to identify the most influential features and prioritize them for further investigation or refinement.
- Incorporate feedback loops and mechanisms for collecting user feedback to iteratively improve the model based on real-world observations and user experiences.

7. Phase 6: Operationalize

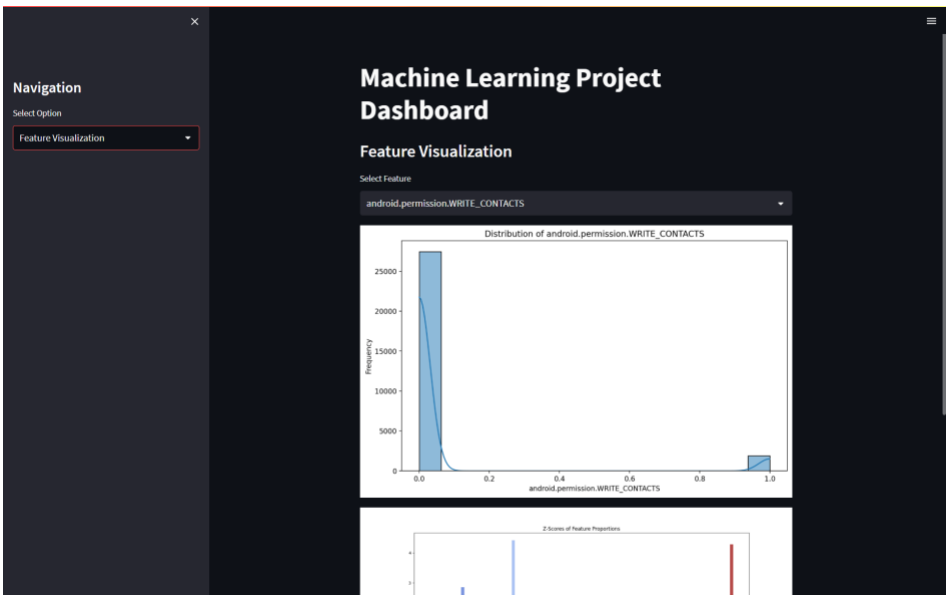
7.1 Steps:

1. **Model Serialization:** Serialize the machine learning model into a compatible format for easy loading and saving, ensuring compatibility with Streamlit.
2. **Streamlit Development:** Utilize Streamlit to create the frontend application. Incorporate interactive components like input fields and buttons to gather user input regarding permission requirements of Android applications.
3. **Integration with Machine Learning Model:** Integrate the serialized machine learning model into the Streamlit application. Define the logic to pass user input data through the model and display the model's predictions.
4. **Visualizations:** Implement visualizations within the Streamlit app to offer insights into the data or model behavior. For instance, visualize feature distributions or model performance metrics.
5. **Testing:** Thoroughly test the Streamlit application to ensure proper functionality. Verify that user inputs are processed accurately and model predictions are reliable.
6. **Deployment:** Deploy the Streamlit application to a suitable hosting platform, such as self-hosting, cloud platforms, or services like Streamlit Sharing. Ensure the deployment environment supports Streamlit applications and the necessary dependencies.
7. **Scalability:** Consider potential scalability requirements for the deployed application. While Streamlit itself doesn't provide built-in scaling features, hosting platforms may offer scalability options. Assess scalability needs and select an appropriate hosting solution.
8. **Security:** Implement necessary security measures to safeguard the deployed application. This may involve data encryption, access control, and protection against common security threats like SQL injection or cross-site scripting (XSS).

7.2 Code Explanation:

- **Importing Libraries:** The code starts by importing necessary libraries such as Streamlit for building the web application, pandas for data manipulation, and matplotlib.pyplot and seaborn for data visualization.
- **Loading Data:** The load_data function, decorated with @st.cache data, loads CSV data from the specified file path into a DataFrame using pd.read_csv. Caching the data with @st.cache data ensures it's loaded only once and cached for subsequent use, improving performance.
- **Visualizing Features:** The visualize_features function takes a DataFrame (df) and a feature name (feature) as input. It creates a Matplotlib figure and axis objects for plotting, visualizing the feature's distribution using sns.histplot (from Seaborn). Finally, the plot is displayed in the Streamlit app using st.pyplot(fig).
- **Visualizing Outliers:** The visualize_outliers function is similar to visualize_features but focuses on visualizing outliers using a boxplot (sns.boxplot). It also creates Matplotlib figure and axis objects, sets titles and labels, and displays the plot using st.pyplot(fig).
- **Main Function:** The main function serves as the entry point of the Streamlit application. It sets the title of the web application and the sidebar title for navigation. Data is loaded using the load_data function and stored in the DataFrame df. Sidebar options are provided using st.sidebar.selectbox. Depending on the selected option, either feature visualization or outlier visualization is displayed. The selected feature is visualized using the appropriate function, and visualization images are displayed using st.image.

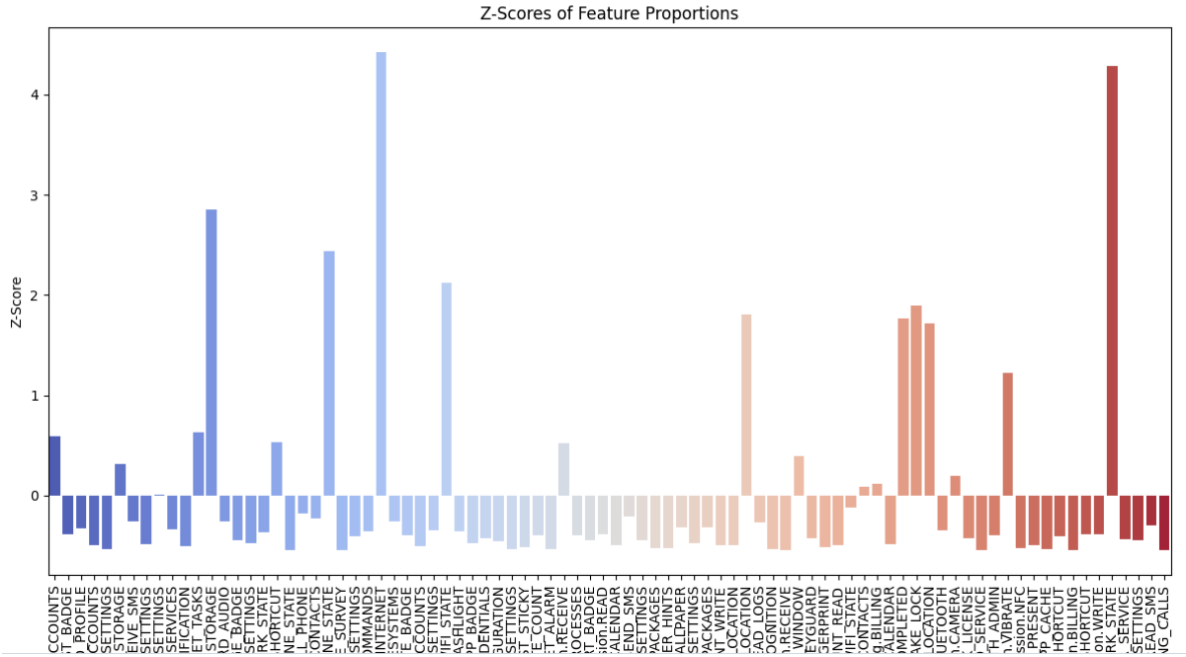
7.3 Web View:



8. Source Code:

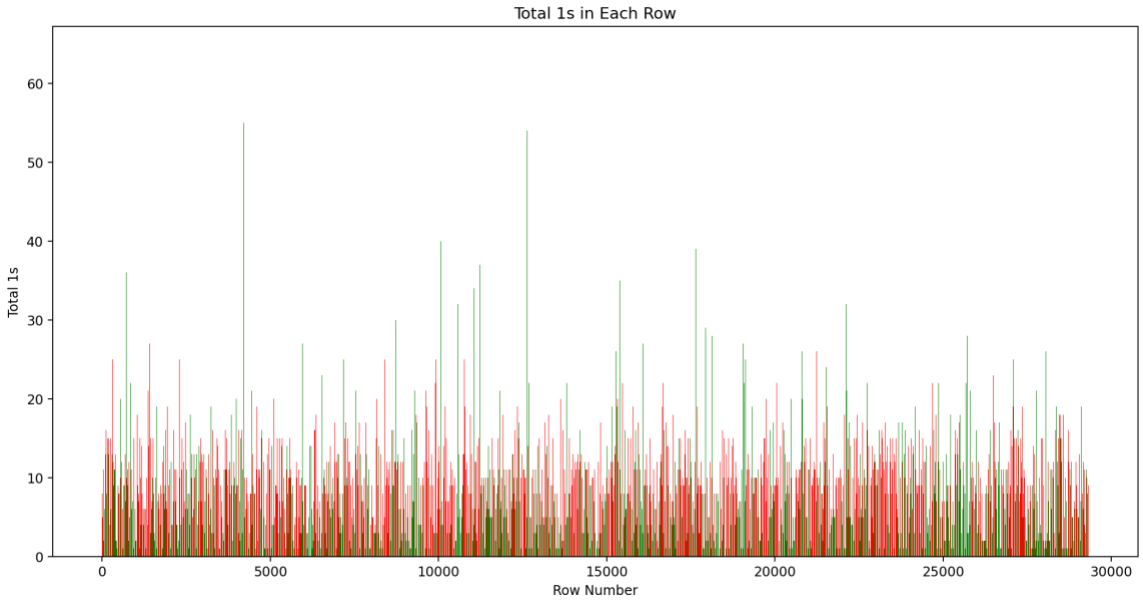
FeatureVisualization.py:

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

data = pd.read_csv('train.csv')

X = data.drop('Result', axis=1)
y = data['Result']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Calculate the proportion of 'Yes' values for each feature
proportions = X.mean()

# Calculate the z-score for each feature
z_scores = (proportions - proportions.mean()) / proportions.std()

# Visualize the z-scores
plt.figure(figsize=(10, 6))
sns.barplot(x=z_scores.index, y=z_scores.values, palette='coolwarm')
plt.title('Z-Scores of Feature Proportions')
plt.xlabel('Feature')
plt.ylabel('Z-Score')
plt.xticks(rotation=90)
plt.show()

# Identify features with z-scores greater than a threshold (e.g., 2)
outliers = z_scores[z_scores.abs() > 2]

print('Features with z-scores greater than 2 (potential outliers):')
print(outliers)
```



OutlierVisualization.py:

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load the CSV file into a pandas DataFrame
df = pd.read_csv('train.csv')
df = df.sample(frac=1).reset_index(drop=True)
# Calculate the total number of 1s in each row
df['total_ones'] = df.sum(axis=1)

# Plot the graph
for index, row in df.iterrows():
    color = 'green' if row['Result'] == 0 else 'red'
    plt.bar(index, row['total_ones'], color=color)

plt.xlabel('Row Number')
plt.ylabel('Total 1s')
plt.title('Total 1s in Each Row')
plt.show()
```



Source.py:

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

data = pd.read_csv('train.csv')

X = data.drop('Result', axis=1)
y = data['Result']

# Normalization
scaler = StandardScaler()
X_normalized = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_normalized, y, test_size=0.2, random_state=42)

model = Sequential([
    Dense(84, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(42, activation='relu'),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test accuracy: {accuracy:.4f}')
```

9. Conclusion:

In the initial phases of Data Discovery and Data Collection and Preparation, the project establishes hypotheses, identifies data sources, explores the business domain, and prepares the dataset for analysis. These steps lay the foundation for understanding the problem domain and gathering the necessary data.

The subsequent phases of Model Planning and Model Building focus on developing appropriate machine learning models for malware detection. Feature selection techniques are employed to refine the dataset, and a neural network model is chosen for its suitability in handling binary classification tasks with non-continuous data. The training and testing results indicate a high level of accuracy, validating the effectiveness of the chosen model.

In the final phase of Operationalize, the project operationalizes the model by integrating it into a Streamlit application. This allows users to interactively input permission requirements for Android applications and receive predictions regarding potential malware presence. The application is thoroughly tested and deployed to ensure proper functionality, scalability, and security.

Overall, the project demonstrates a structured and methodical approach to addressing the problem of malware detection in Android applications. By leveraging machine learning techniques and developing a user-friendly application, the project aims to enhance security measures and protect users from potential threats in the mobile ecosystem.