DIGI GUARDIAN : MALWARE DETECTION

ABSTRACT

The increasing reliance on mobile devices in our daily lives has given rise to a surge in malware attacks, posing significant threats to personal data and security. In response to this challenge, our project, "Digi Guardian: Malware Detection," aims to develop a robust system capable of efficiently detecting and classifying malware across various devices, including mobile and IoT systems.

The project follows a structured methodology encompassing stages starting from data analysis till machine learning model development. The process begins with data pre-processing, where missing values are addressed, categorical variables are encoded using one-hot encoding, and numerical variables are standardized through normalization. Subsequently, data visualization is employed to explore the dataset visually, identifying anomalies such as outliers and discerning trends. Feature selection follows, involving the extraction and categorization of relevant features to streamline the model.

Moving to the model training phase, the data is divided into training, validation and test sets, and algorithms like Logistic Regression, Naive Bayes, SVM and Neural Networks are employed to train and test the model. Model Evaluation is a critical step, assessing the performance factors like precision, accuracy, recall, and F1-Score. The confusion matrix is utilized to analyze the model's predictions. Finally, the developed model is tested on new data, enabling a comprehensive evaluation of its real-world performance. This systematic approach ensures a robust and reliable outcome for the project.

INTRODUCTION:

In today's tech-heavy world, our cell phones and tablets are crucial. They weave into each part of our daily lives. But, this reliance puts users at risk of harmful computer viruses. Frequently, users unknowingly let apps have too much access when they are set up. This open door lets harmful software attack. This project looks at an urgent requirement for a system that can combat such hazards. The aim is to keep our digital surroundings safe by making a system that can separate apps into two groups - safe or dangerous - based on the access they ask for.

PROBLEM DEFINITION:

Mobile devices have become vital to everyone's lifestyle. It is hard to imagine one's day without them. However, because we rely on mobile phones so heavily, certain malware attacks have been rapidly growing. A user tends to download an application from the app store for various reasons. Most of the time we neglect to check the permission settings that are granted for an application in default and proceed ahead without thinking twice. Certain default permissions turn into a path that leads to malware attacks on our devices. Malicious software gets downloaded and thereby causes significant risks. This malware, once present, poses significant threats to personal data and security. As hackers continue to exploit various vulnerabilities in IoT systems, the seamless integration of IoT technology into our lives is at risk.

To tackle these increasingly complex challenges, there is a pressing demand for the development of a resilient system designed to proficiently detect and classify malware across a spectrum of devices, encompassing both mobile and IoT systems. To address these converging challenges, the need arises for a robust system capable of detecting malware on mobile devices, ensuring a secure digital environment for all. The model being developed must be able to classify whether the data is Malware i.e., our class of interest, or safe based on the permissions as requested.

DATA SOURCES:

We have chosen our dataset from the UCI repository. The link for the dataset is: https://archive.ics.uci.edu/dataset/722/naticusdroid+android+permissions+dataset

DATA DESCRIPTION:

The chosen dataset has 86 distinct features and 29333 instances, with two outcomes; which can be used to train and test our model. Each column represents a specific permission granted to that respective application.

Some of the features are as the following:

- Read Access: Access to read the data present
- Write Access: Access to edit the data such as user preferences
- External Storage Access: Access to External Storage
- Cache Memory Access: Access to cache memory
- Receive SMS: Access to message alerts
- Manage Accounts: Whether multiple accounts data needs to be saved
- Location Access: Location of the user must be available to the application
- Install Shortcuts: Access to create shortcuts on the homepage of the device
- Bluetooth Access: Whether Bluetooth needs to be enabled always or not
- Camera Access: Whether camera access must be provided to that application
- Billing Access: Whether credit card or debit card details need to be saved
- Authentication Access: Access to verify the user's identity

The dataset consists of binary values either 0 or 1. A binary value 0 indicates that permission has not been requested while value 1 indicates that permission is required for that application.

The Result column denotes the presence or absence of malware. In the future, we can anticipate whether the application is malicious or not based on the permissions requested.

EXPLORATORY DATA ANALYSIS AND FEATURE ENGINEERING:

Univariate Analysis:

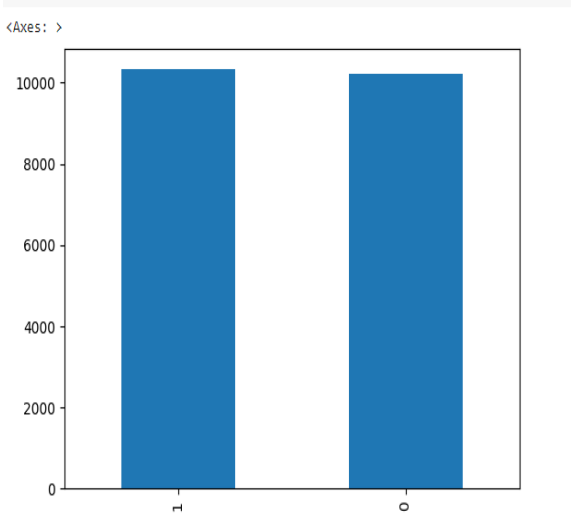
Univariate analysis is a statistical approach focused on the examination of a single variable at a time. This step is crucial to identify the outliers and missing values within a single feature. This helps to clean the data to enhance the data quality for further modeling. The insights gained from the univariate analysis are:

- The dataset has no null values.
- The entire dataset is in binary format i.e., int64. Hence, standardization is not required.

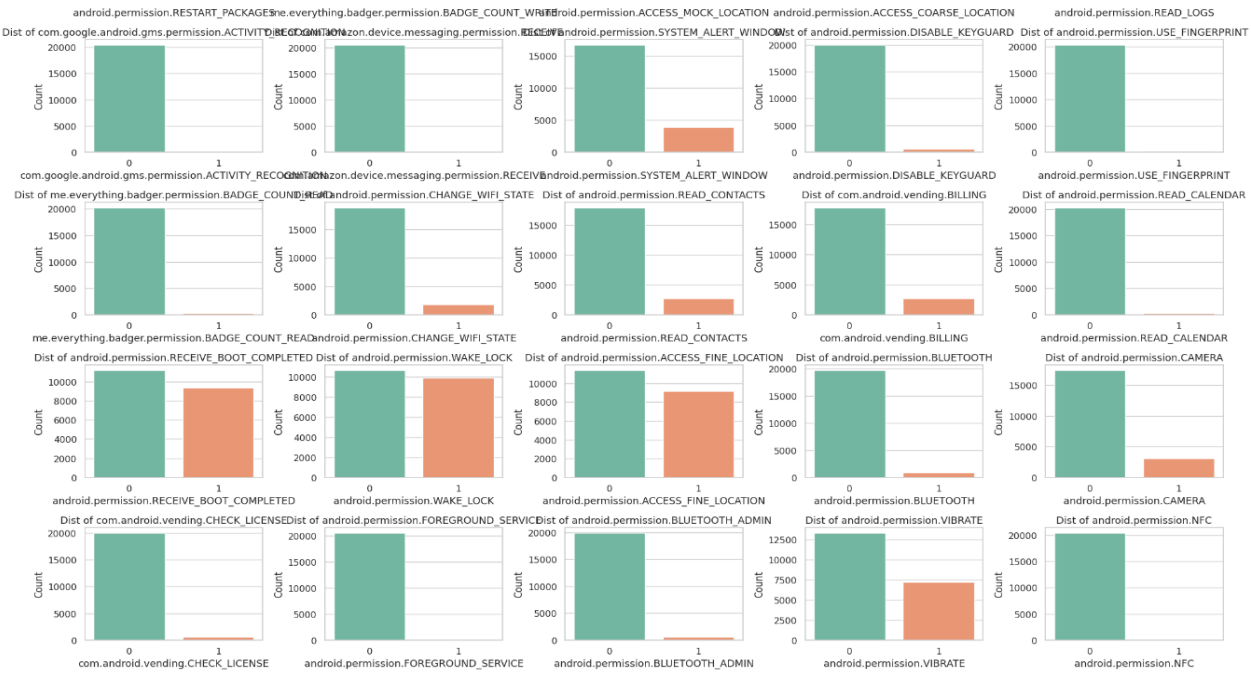
	ColumnName	#ofNullValues	ColumnType	#OfUniqueValues	SampleData
0	android.permission.GET_ACCOUNTS	0	int64	2	[0, 1]
1	com.sonyericsson.home.permission.BROADCAST_BADGE	0	int64	2	[0, 1]
2	android.permission.READ_PROFILE	0	int64	2	[0, 1]
3	android.permission.MANAGE_ACCOUNTS	0	int64	2	[0, 1]
4	android.permission.WRITE_SYNC_SETTINGS	0	int64	2	[0, 1]
5	android.permission.READ_EXTERNAL_STORAGE	0	int64	2	[0, 1]
6	android.permission.RECEIVE_SMS	0	int64	2	[0, 1]
7	com.android.launcher.permission.READ_SETTINGS	0	int64	2	[0, 1]
8	android.permission.WRITE_SETTINGS	0	int64	2	[0, 1]
9	com.google.android.providers.gsf.permission.RE...	0	int64	2	[0, 1]
10	android.permission.DOWNLOAD_WITHOUT_NOTIFICATION	0	int64	2	[0, 1]
11	android.permission.GET_TASKS	0	int64	2	[0, 1]
12	android.permission.WRITE_EXTERNAL_STORAGE	0	int64	2	[0, 1]
13	android.permission.RECORD_AUDIO	0	int64	2	[0, 1]
14	com.huawei.android.launcher.permission.CHANGE_...	0	int64	2	[0, 1]
15	com.oppo.launcher.permission.READ_SETTINGS	0	int64	2	[0, 1]
16	android.permission.CHANGE_NETWORK_STATE	0	int64	2	[0, 1]
17	com.android.launcher.permission.INSTALL_SHORTCUT	0	int64	2	[0, 1]
18	android.permission.android.permission.READ_PHO...	0	int64	2	[0, 1]
19	android.permission.CALL_PHONE	0	int64	2	[0, 1]
20	android.permission.WRITE_CONTACTS	0	int64	2	[0, 1]

Distribution Analysis:

Distribution Analysis is a process of understanding the distribution of values within a dataset. It helps in identifying the patterns and statistical properties of the data to understand its behavior.



The graph indicates the distribution behavior of the target variable which is the **Result** column. It can be observed that it is equally distributed.
Hence, we try to understand the distribution behavior of the remaining features.



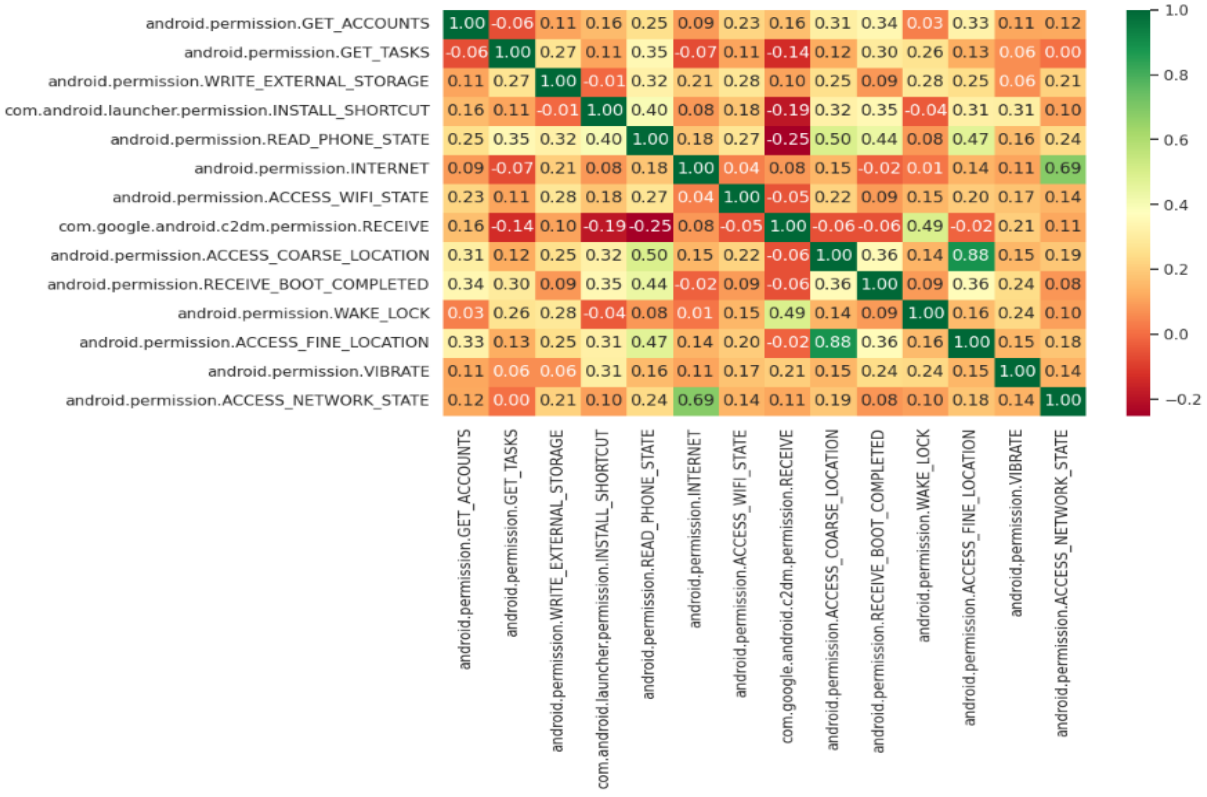
The above plot is a snip of the distribution analysis plot. It can be interpreted that most of the columns are highly imbalanced. Therefore, we need to drop those columns to proceed ahead. Nearly 76 columns of unbalanced data have been dropped.

Now, we proceed ahead with the remaining 14 columns and perform correlation analysis.

Correlation Analysis:

Correlation Analysis can be defined as a statistical technique which is used to evaluate the strength and direction of the linear relationship between any two quantitative variables.
It helps to identify the similarities and trends in the data. The correlation coefficients give us how strong and what is the direction of linear relationship between any two features.
The range of correlation coefficient lies between -1 and 1.

- Value of 1 represents Perfect Positive Correlation
- Value of 0 suggests No Linear Relationship between features
- Value of -1 represents Perfect Negative Correlation



After retrieving the correlation heatmap, we drop highly correlated features from our data. We assigned a threshold value of 0.8. Any feature with a threshold value greater than 0.8 has been dropped.

Mutual Information:

Mutual Information is a statistical metric that gives us a measure of the amount of information shared between two variables.

It is frequently used for feature selection. Unlike our correlation matrix, this identifies both linear and non-linear associations between variables.

Features with high mutual information with the target variable are considered more informative and may be prioritized in the modeling process. A low mutual information score suggests that its not impacting the target variable.

As shown below, MI value for "WAKE_LOCK" is zero which means its not impacting the target variable. So, we are removing that feature.

```
android.permission.READ_PHONE_STATE                 0.298949
com.google.android.c2dm.permission.RECEIVE          0.142591
android.permission.RECEIVE_BOOT_COMPLETED           0.128006
com.android.launcher.permission.INSTALL_SHORTCUT    0.115834
android.permission.ACCESS_COARSE_LOCATION           0.097542
android.permission.GET_TASKS                        0.083415
android.permission.ACCESS_WIFI_STATE                0.025880
android.permission.WRITE_EXTERNAL_STORAGE           0.011036
android.permission.GET_ACCOUNTS                     0.010047
android.permission.ACCESS_NETWORK_STATE             0.009643
android.permission.INTERNET                         0.004147
android.permission.VIBRATE                          0.002703
android.permission.WAKE_LOCK                        0.000000
dtype: float64
```

SPLITING THE DATA FOR MODEL TRANING:

1. The dataset was divided into three parts: training (70%), validation (15%), and test (15%).
2. The test set is kept separate, ensuring that the model doesn't see this data during training and evaluation.
3. Evaluate the model's performance on the validation set using metrics such as accuracy, precision, recall.

4. Tune the model hyperparameters using techniques to improve the model's performance.

MODEL SELECTION & IMPLEMENTATION:

1. Logistic Regression:

Logistic Regression is a classification technique that works optimally when the data is binary, linearly independent and doesn't have any outliers. Logistic Regression generates a binary output by utilizing the sigmoid function.

The steps involved are:

1. Initialization:

The class LogisticRegression is initialized with training and testing data (train_X, test_X, train_y, test_y), learning rate (learning_rate), maximum iterations for gradient descent (max_iterations), and a convergence threshold (epsilon).

2. Adding Bias Term:

The method add_X0 is used to add a bias term to the input features. It adds a column of ones to the input matrix.

3. Sigmoid Function:

The method sigmoid computes the sigmoid function for a given input.

```
def sigmiod(self,X):
  sig = 1/(1+np.exp(-X.dot(self.theta)))
  return sig
```

4. Prediction:

The predict method uses the sigmoid function to predict binary outcomes (0 or 1) based on the input features.

5. Cost Function:

The cost_function method calculates the logistic regression cost function, which represents the difference between predicted and actual values.

```
def cost_function(self,X,y):
  sig=self.sigmiod(X)
  pred=(1/X.shape[0]) * ((y * np.log(sig)) + ((1-y) * np.log(1-sig)))
  cost=-pred.sum()
  return cost
```

6. Cost Derivative:

The cost_derivative method computes the gradient of the cost function with respect to the parameters (theta).

```
def cost_derivative(self, X, y):
  sig = self.sigmiod(X)
  grad = (sig - y).dot(X)
  return grad
```

7. Gradient Descent:

The gradient_descent method performs gradient descent to optimize the model parameters (theta) based on the training data. It iteratively updates theta to minimize the cost function.

8. Fitting the Model:

The fit method initializes the model parameters, performs gradient descent on the training data, and prints training and testing accuracy, precision, and recall.

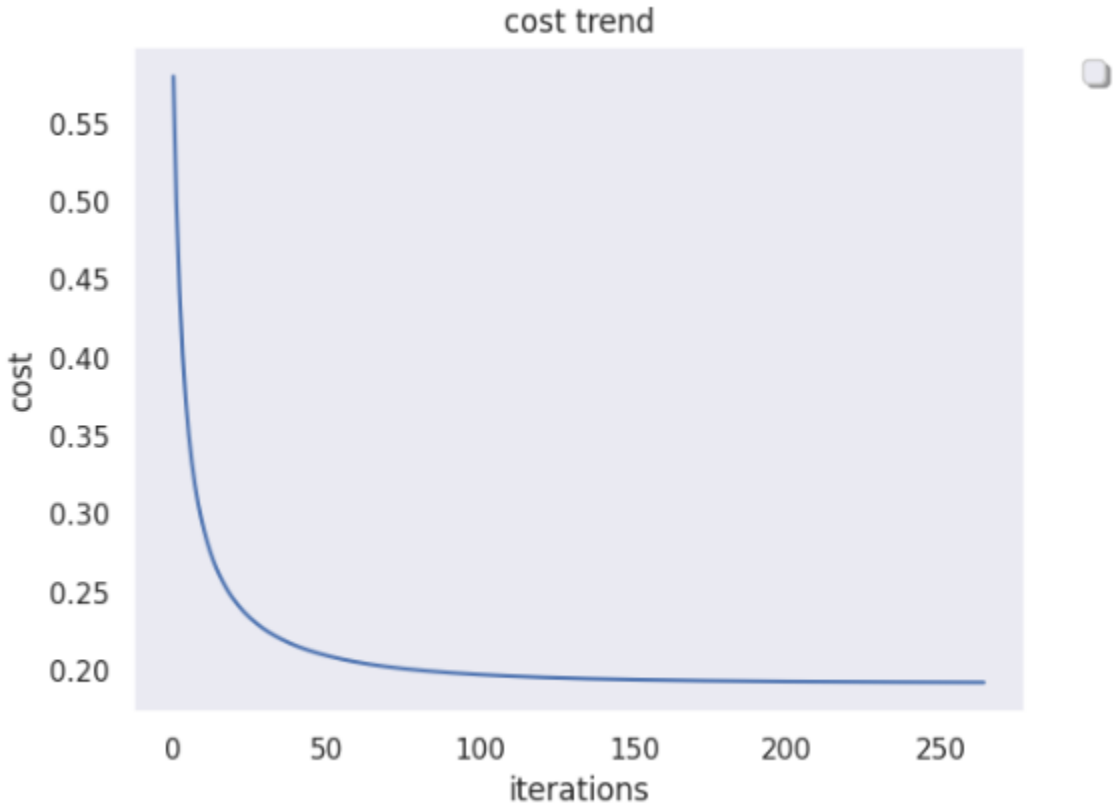
9. Performance Metrics:

The matrix method calculates accuracy, precision, and recall based on the predicted and actual values.

10. Plotting Cost Trend:

The plot Cost method visualizes the trend of the cost function over iterations using Matplotlib.

Upon training the model, we get the following results:



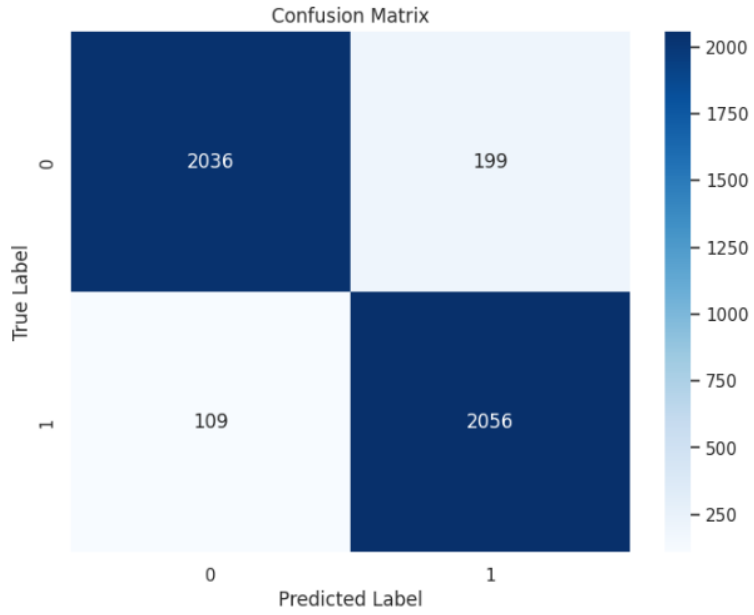
In the visual representation depicted in the figure, it is evident that our model demonstrates a smooth convergence pattern. Notably, the learning process halts gracefully after 250 iterations with a learning rate of 0.0001. In our pursuit of optimizing the model's performance, we systematically experimented with various hyperparameter combinations. Through rigorous testing, we identified and selected the most effective set of hyperparameters, resulting in optimal model outcomes.

Evaluation Metrics:

The values of accuracy and precision for train and test data are:

Metrics	**Train**	**Test**
Accuracy	0.9247	0.9195
Precision	0.9028	0.8928
Recall	0.9528	0.9506

Confusion Matrix:



The examination of the confusion matrix reveals that the accuracy metrics for both the validation and training sets exhibit a noteworthy similarity. This observation serves as a strong indicator that our model is not overfitting, thereby affirming its generalization capability across different datasets.

Feature Importance:

Feature Importance	
com.google.android.c2dm.permission.RECEIVE	-4.628765
android.permission.INTERNET	-1.844326
android.permission.ACCESS_NETWORK_STATE	-1.124051
android.permission.VIBRATE	-0.86671
android.permission.WRITE_EXTERNAL_STORAGE	-0.837778
android.permission.ACCESS_WIFI_STATE	-0.035333
android.permission.ACCESS_COARSE_LOCATION	1.162467
Theta android.permission.GET_TASKS	1.3545
android.permission.RECEIVE_BOOT_COMPLETED	1.514481
com.android.launcher.permission.INSTALL_SHORTCUT	1.861691
android.permission.READ_PHONE_STATE	4.133827

Model Interpretation:

1. Feature importance helps in understanding how each feature influences the target variable by looking at the model coefficients.
2. The model's predictions using tools like the confusion matrix and precision-recall curve to understand how the model behaves.

2. <u>Naive Bayes:</u>

Naive Bayes belongs to a family of probabilistic algorithms based on Bayes' theorem. Here, we assume that the features are identically independent. It is particularly suitable for text classification tasks like spam filtering and sentiment analysis, as well as document categorization.

Naive Bayes is computationally efficient, making it suitable for quick classification tasks with small datasets. It performs well in scenarios where the independence assumption holds and is effective for multiclass classification.

The steps involved are:

1. Initialization:

The NaiveBayes class is initialized with training data (X_train, y_train) and a smoothing parameter (tau), which is set to 1.0 by default.

2. __post_init__ Method:

The __post_init__ method is a special method called automatically after the object is created. In this method, the fit method is called to fit the model to the training data.

3. Likelihood Calculation:

The likelihood method calculates the probabilities for all possibilities for each class with Laplace Smoothing. It returns a dictionary (prob) containing probabilities for each column of the input data.

```
def likelihood(self, data):
  '''
  Calculating the probabilities for all possibilities for each class
  with Laplace Smoothing
  '''
  prob = {}
  for col in range(data.shape[1]):
    temp = data[:,col]
    prob[col] = {}
    for var in set(temp):
      prob[col][var] = (sum(temp == var) + self.tau)/ (temp.shape[0] + (self.tau * len(set(temp))))
  return prob
```

4. Fit Method:

The fit method fits the model to the training data. It calculates class priors and likelihoods for each class based on the training data.

5. Probability Zero Calculation:

The prob_zero method calculates Laplace smoothing if the probability is zero.

```
def prob_zero(self,col,class_):
  '''
  Calculating Laplace if the probability is Zero
  '''
  temp = self.X_train[self.y_train == class_][:, col]
  value = self.tau / (sum(self.y_train == class_) + (self.tau + len(set(temp))))
  return value
```

6. Predict Method:

The predict method predicts the class labels for the test data. It calculates the probabilities for each class and makes predictions based on the maximum probability.

7. Confusion Matrix Method:

The confusion_matrix method calculates and prints accuracy, precision, and recall based on the predicted and actual values.

The values of accuracy and precision are:

Metrics	**Train**	**Test**
Accuracy	0.901	0.906
Precision	0.914	0.911
Recall	0.887	0.896

As anticipated, Naive Bayes exhibited superior performance in binary classification when applied to features assumed to be independent (given the correlation between features being less than 20%), achieving an accuracy of 90%. Nevertheless, it did not outperform logistic regression, suggesting that logistic regression might excel in capturing relationships between variables.

3. Support Vector Machine:

SVM models are used for classification and regression models. These models find the hyperplane that best separates data points into different classes in a high-dimensional space. SVMs are highly effective for datasets where the decision boundary is complex and can handle high-dimensional spaces well, making them suitable for binary classification.

The steps involved are:

1. Initialization:

__init__ method initializes the SVM with hyperparameters: lambda_ (regularization parameter), lr (learning rate), and max_iteration (maximum iterations for training).

2. Fit Method:

The SVM's fit method trains the model on input data (X) and labels (y), initializing the weight vector (self.w) and bias (self.b). It converts the target variable to -1 and 1 (y_) and iterates through the dataset for a specified number of iterations. For each sample, it checks if the condition for margin and correct classification is met, updating the weight and bias using gradient descent.

```
y_ = np.where(y <= 0, -1, 1)

for _ in tqdm(range(self.max_iteration), colour='blue'):
    # Loop through each sample in the dataset
    for idx, x_i in enumerate(X):
        # Check the condition for the margin and correct classification
        condition = y_[idx] * (np.dot(x_i, self.w) + self.b) >= 1


        if condition:
            dw = 2 * self.lambda_ * self.w
            db = 0
        else:
            dw = 2 * self.lambda_ * self.w - np.dot(x_i, y_[idx])
            db = -y_[idx]
        self.w -= self.lr * dw
        self.b -= self.lr * db
```

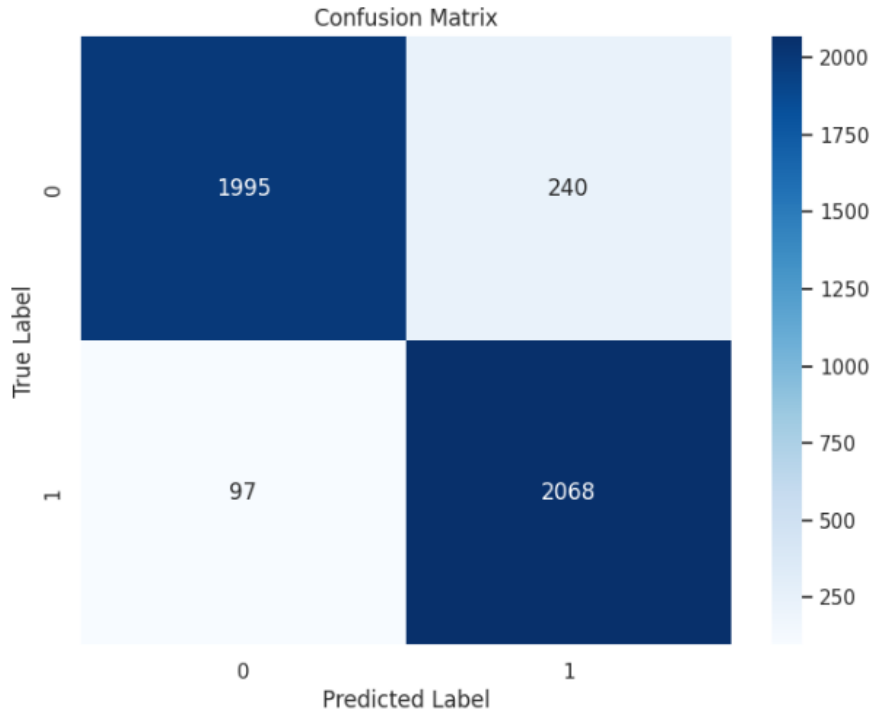
3. Predict Method:

The predict method generates predictions on new data (X) and compares them with true labels (y_true), computing various metrics including True Positives, True Negatives, False Positives, and False Negatives. It further calculates metrics such as accuracy, precision, recall, and F1 score, and visualizes the confusion matrix using Matplotlib and Seaborn.

The values of accuracy and precision for train and test data are:

Metrics	**Train**	**Test**
Accuracy	0.9222	0.9234
Precision	0.8933	0.8960
Recall	0.9598	0..9552

Confusion Matrix:



Upon examining the metrics of the Support Vector Machine (SVM), it is evident that its performance aligns closely with logistic regression. This observation could be attributed to the possibility that SVM might demonstrate superior performance in the presence of

nonlinear relations between features. However, it appears that the data is linearly separable, favoring logistic regression over SVM in this specific scenario.

4. Neural Networks:

Neural networks are a class of machine learning models inspired by the structure and functioning of the human brain. They consist of interconnected nodes, or artificial neurons, organized into layers. These are used when the data has non-linearities that a simpler model like Logistic Regression may not capture.

The steps involved are:

1. Initialization:

The Tensor Flow and Keras Libraries are being imported and a sequential model, which is a stack of layers is being created.

2. Adding Layers to the Model:

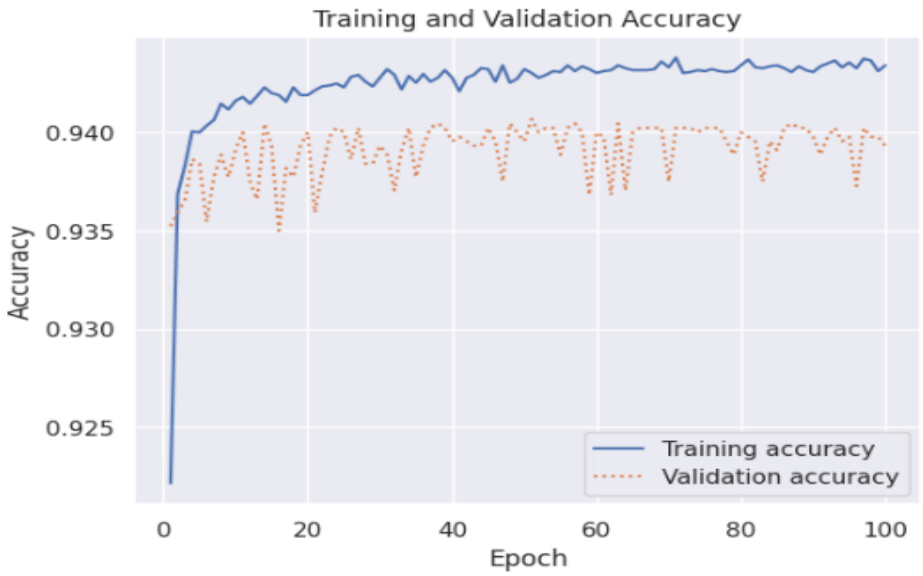
```
model.add(Dense(units=len(features), input_dim=len(features), activation='relu'))
model.add(Dense(512,activation='relu'))
model.add(Dense(256,activation='relu'))
# model.add(Dense(128,activation='relu'))
model.add(Dense(units=1, activation='sigmoid'))
```

The first layer (input layer)has been added with a number of units equal to the length of the features. This layer uses the Rectified Linear Unit (ReLU) activation function. The two hidden layers with 512 and 256 units, have been added both using the ReLU activation function. The output layer with 1 unit and a sigmoid activation function has been added. The sigmoid activation is commonly used for binary classification tasks

3. Compilation:

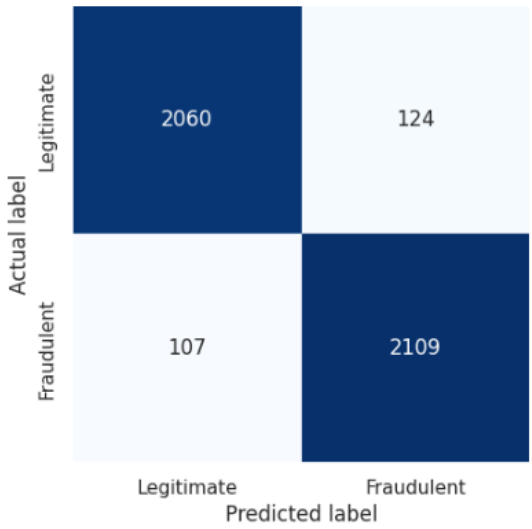
Finally, the model has been compiled with Adam Optimizer and accuracy has been used as an evaluation metric.

[ ]



Interpreting the Training and Validation Accuracy plot for neural networks involves observing the convergence of accuracy values over training epochs. As we can see, both curves are increasing and eventually getting plateau, indicating effective learning.

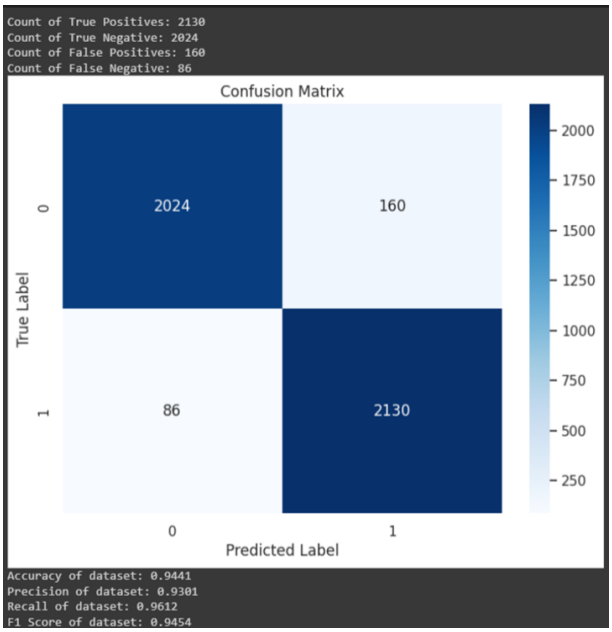
Confusion Matrix:



Generalization & Bias Variance Trade-Off:

Generalization refers to a model's proficiency in delivering good performance on new and previously unseen data. Achieving this requires careful data splitting, dividing the dataset into training, validation, and test sets for robust model evaluation. The final model is rigorously assessed on the test set to gauge its generalization capabilities.

The test set's performance serves as a crucial benchmark, reflecting how well the model adapts to entirely unfamiliar data. The model's alignment with training and validation accuracies suggests a lack of overfitting to the training data. Notably, the higher accuracy observed on the test set (94%) is an encouraging indication of the model's strong generalization to novel and unseen instances. Test accuracy stands out as a vital metric for appraising the model's effectiveness in real-world scenarios.

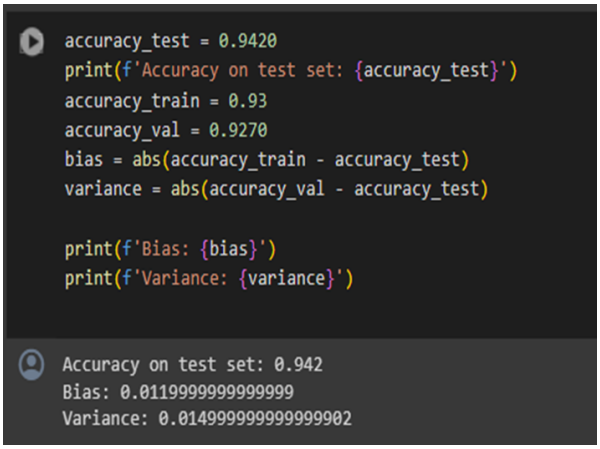


Bias and Variance:

- **Bias (0.012):** The bias measures the difference between the training accuracy and test accuracy. A bias of approximately 1.2% suggests a slight underestimation of the model's performance on the test set compared to the training set.
- **Variance (0.015):** The variance measures the difference between the validation accuracy and test accuracy. A variance of approximately 1.5% suggests a slight variability in the model's performance across different subsets of data.

Insights:

The bias and variance values are relatively small, which is a positive indication of the model's stability and generalization.



```
accuracy_test = 0.9420
print(f'Accuracy on test set: {accuracy_test}')
accuracy_train = 0.93
accuracy_val = 0.9270
bias = abs(accuracy_train - accuracy_test)
variance = abs(accuracy_val - accuracy_test)

print(f'Bias: {bias}')
print(f'Variance: {variance}')
```

```
Accuracy on test set: 0.942
Bias: 0.0119999999999999
Variance: 0.014999999999999902
```

Results & Conclusion:

It's time to choose our winning model, hence we have to compare the overall model performance based on the performance metrics.

Metric	Logistic Regression	Naive Bayes	SVM	Neural Network
Accuracy	0.9300	0.9061	0.9234	0.9500
Precision	0.9118	0.9112	0..8960	0.9432
Recall	0.9497	0.8965	0.9522	0.9506
Hyperparameters	Learning Rate=0.0001	Tau = 1.0	Lambda= 0.001 Learning Rate=0.0001	Epochs= 100 Batch Size= 32
Time Taken	9.3 microseconds	8.58 microseconds	1.45 mins	5min

The selection of Logistic Regression as our winning model is based on several considerations:

Suitability for Binary Outcomes:

Logistic Regression was initially chosen for its appropriateness in handling binary outcomes, making it well-suited for our classification task.

Linearly Separable Data:

The model's effectiveness is enhanced by its compatibility with linearly separable data, contributing to its robust performance.

Independence and Outlier Handling:

Logistic Regression is adept at handling independent data with no outliers, aligning well with the characteristics of our dataset.

Interpretability:

The model's provision of clear interpretability through coefficients adds a valuable layer of transparency to the analysis.

Simplicity as a Baseline:

Logistic Regression's simplicity positions it favorably as a baseline model, facilitating a straightforward understanding of the underlying relationships.

Consideration of Model Complexity:

Despite Neural Networks demonstrating the highest accuracy, the marginal improvement may not warrant the increased complexity associated with binary classification and binary features.

In summary, Logistic Regression emerges as the best choice, aligning with the nature of the data and balancing interpretability, simplicity, and performance in the context of our specific classification task.