ASSESSMENT TASK REPORT

FDA_A3

32130_AT2_24999031

Rubric#1; Data Mining Problem:

Creating a classification model to identify different types of cyber intrusion in an Internet of Things (IoT) network is the issue at hand. Because of their widespread use and sometimes insufficient security setups, Internet of Things (IoT) devices—which include anything from smart appliances to industrial control systems—are becoming more and more susceptible to cyberattacks.

The study's dataset contains network traffic data, with each instance representing a network flow that was recorded and identified by a number of characteristics, including protocol type, TCP flags, and flow length. Each flow's class is represented by the target attribute name, which indicates if the flow is part of the DictionaryBruteForce, Recon-OSScan, or Mirai-greip_flood assault types.

The classification problem can be viewed as a multi-class classification task, where the goal is to accurately classify these network traffic instances into their respective categories.

This classification problem is vital in the context of **intrusion detection systems (IDS)**, which are designed to identify and respond to cyber threats in real-time. By training machine learning models on labeled data, we aim to automate the detection of these intrusions and improve the overall security posture of IoT networks. In this project, various machine learning techniques will be applied to identify the most effective approach for this classification problem.

Rubric #2: Data Pre-processing and Transformation (9 pts)

Data pre-processing is a crucial step in the data mining process as it ensures that the dataset is clean, relevant, and suitable for the chosen classification algorithms. For this task, the dataset required several pre-processing steps, including handling missing values, removing irrelevant or redundant features, transforming categorical data, and addressing imbalanced class distributions.

1. **Handling Missing Data:**
   All rows in the dataset were complete, meaning no imputation was necessary. This helped streamline the pre-processing, allowing focus to shift toward other aspects like feature selection and transformation.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15600 entries, 0 to 15599
Data columns (total 48 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Unnamed: 0       15600 non-null  int64
 1   flow_duration    15600 non-null  float64
 2   Header_Length    15600 non-null  float64
 3   Protocol Type    15600 non-null  float64
 4   Duration         15600 non-null  float64
 5   Rate             15600 non-null  float64
 6   Srate            15600 non-null  float64
 7   Drate            15600 non-null  float64
 8   fin_flag_number  15600 non-null  int64
 9   syn_flag_number  15600 non-null  int64
 10  rst_flag_number  15600 non-null  int64
 11  psh_flag_number  15600 non-null  int64
 12  ack_flag_number  15600 non-null  int64
 13  ece_flag_number  15600 non-null  int64
 14  cwr_flag_number  15600 non-null  int64
 15  ack_count        15600 non-null  float64
 16  syn_count        15600 non-null  float64
 17  fin_count        15600 non-null  float64
 18  urg_count        15600 non-null  float64
 19  rst_count        15600 non-null  float64
 20  HTTP             15600 non-null  int64
 21  HTTPS            15600 non-null  int64
 22  DNS              15600 non-null  int64
 23  Telnet           15600 non-null  int64
 24  SMTP             15600 non-null  int64
 25  SSH              15600 non-null  int64
 26  IRC              15600 non-null  int64
 27  TCP              15600 non-null  int64
 28  UDP              15600 non-null  int64
 29  DHCP             15600 non-null  int64
 30  ARP              15600 non-null  int64
 31  ICMP             15600 non-null  int64
 32  IPv              15600 non-null  int64
 33  LLC              15600 non-null  int64
 34  Tot sum          15600 non-null  float64
 35  Min              15600 non-null  float64
 36  Max              15600 non-null  float64
 37  AVG              15600 non-null  float64
 38  Std              15600 non-null  float64
 39  Tot size         15600 non-null  float64
...
 46  Weight           15600 non-null  float64
 47  label            15600 non-null  string
dtypes: float64(25), int64(22), string(1)
memory usage: 5.7 MB
```

2. Feature Selection and Redundancy Removal

Certain columns were dropped due to their irrelevance or redundancy. For example, the `Unnamed: 0` column, which contained timestamps, was removed as it held no informational value for classification. Other columns like Drate, ece_flag_number, cwr_flag_number, and some protocol-based features such as Telnet, SMTP, IRC, and DHCP were dropped as they had zero variance, meaning they were constant across all observations. These features were not useful for distinguishing between classes and could introduce noise into the model.

```
df.drop(['Unnamed: 0','Drate','ece_flag_number','cwr_flag_number','Telnet','SMTP','IRC','DHCP'], axis=1, inplace=True)
df
#"Unnamed:0" is timestamps unrelated to our data. Rest of all have all zero entries in the dataset. Dropping all these columns
```

✓ 0.0s Python

This selection process is supported by the principle of **dimensionality reduction**, which is vital in improving model performance by removing irrelevant features that do not contribute to the classification task.

3. Handling Outliers

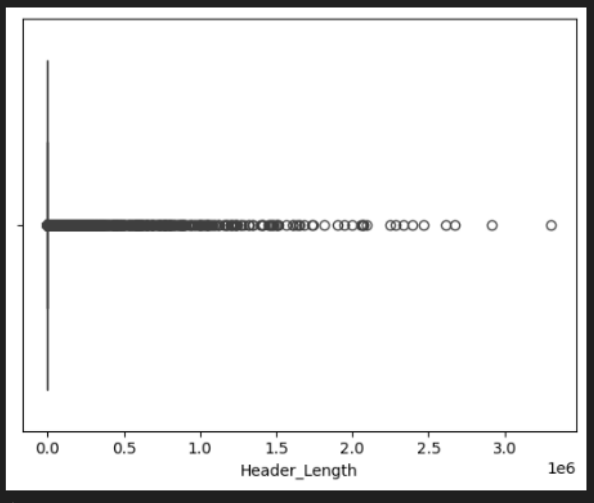
Outlier detection and removal were crucial to improving model accuracy. Outliers can distort the predictions of many classifiers, especially when they involve continuous variables like `flow_duration` and `Header_Length`. Visualizing the distribution of these features using **boxplots** revealed significant outliers.

- **`Header_Length`:** Outliers with values greater than 5,000,000 were removed as they could skew the model.

```
df = df[df['Header_Length'] <= 5000000]
```

```
sns.boxplot(x=df['Header_Length'])
```

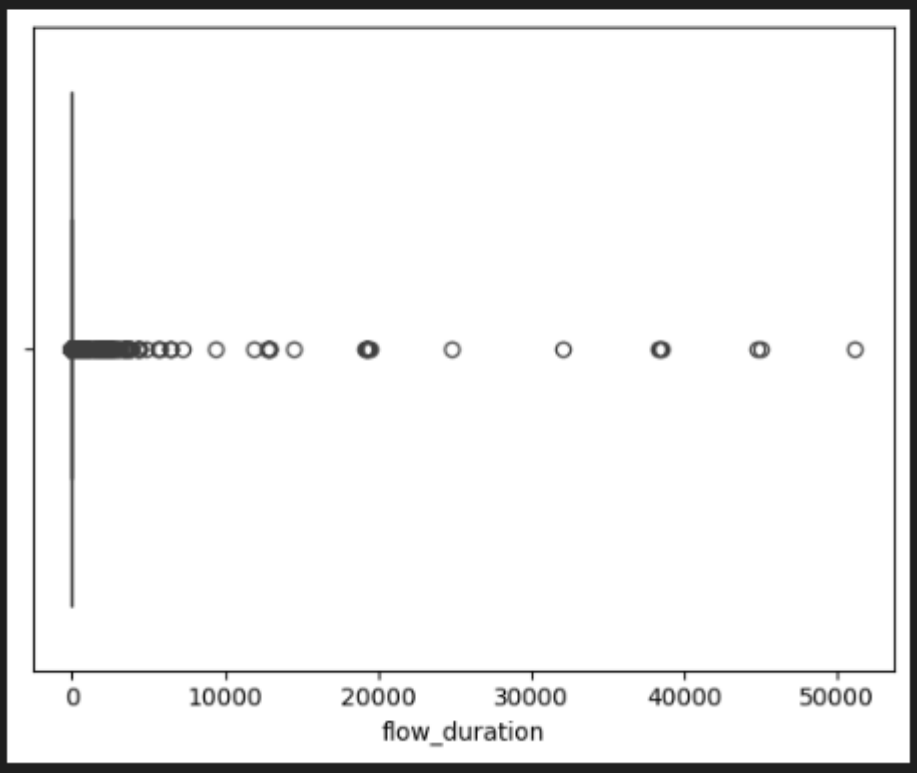
```
<Axes: xlabel='Header_Length'>
```



- **`flow_duration`:** Flows with durations exceeding 10,000 units were identified as outliers and subsequently removed. After removing these outliers, the boxplots indicated a more balanced distribution of these variables, improving model robustness.

```
sns.boxplot(x=df['flow_duration'])
#Outliers detected
```

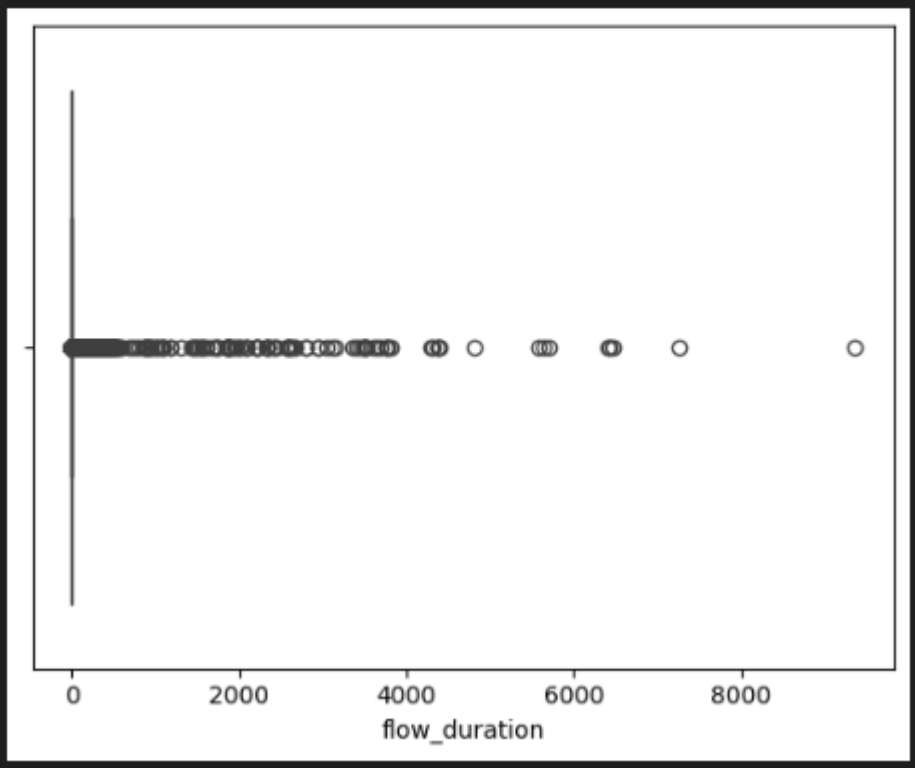
```
<Axes: xlabel='flow_duration'>
```



```
df = df.drop(df[df['flow_duration']>10000].index)
sns.boxplot(x=df['flow_duration'])
#Outliers Removed. Boxplot is balanced
```

✓ 1.3s

```
<Axes: xlabel='flow_duration'>
```



4. Handling Categorical Data

The target variable, `label`, was categorical, representing different types of attacks. Since machine learning algorithms typically require numeric input, the categorical labels were converted into numeric form using **Label Encoding**. This approach replaced each unique class label with a corresponding integer, simplifying the input for the model while maintaining the relationships between categories.

- The following mappings were generated:
  - DictionaryBruteForce → 0
  - Mirai-greip_flood → 1
  - Recon-OSScan → 2

```python
from sklearn.preprocessing import LabelEncoder

# Initialize label encoder
label_encoder = LabelEncoder()

# Fit and transform the label column
df['label'] = label_encoder.fit_transform(df['label'])

# Now, 'label' is numeric (e.g., 0, 1, 2)
```

✓ 5.4s

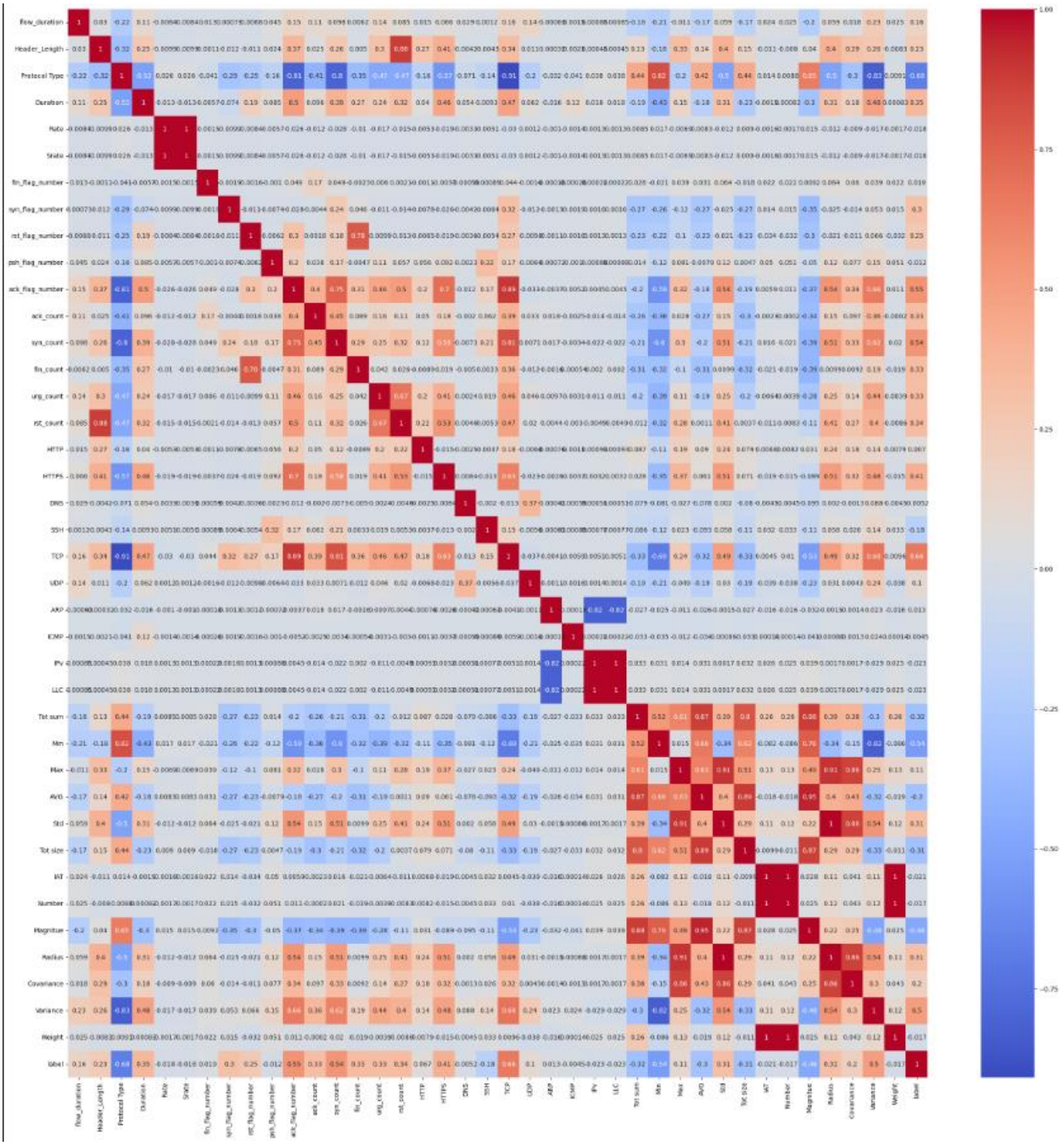
```python
# Check the mappings
label_mappings = dict(zip(label_encoder.classes_, label_encoder.transform(label_encoder.classes_)))
print(label_mappings)
```

✓ 0.0s

```
{'DictionaryBruteForce': np.int64(0), 'Mirai-greip_flood': np.int64(1), 'Recon-OSScan': np.int64(2)}
```

5. Correlation Analysis and Feature Reduction

To further refine the dataset, **correlation analysis** was performed to identify highly correlated features.



Features that exhibited a correlation value of 1.0 were deemed redundant and removed, as they did not provide new information. For example, features such as Srate, LLC, Number, Std, and IAT were removed due to their high correlation with other variables.

```
#Removing all column which have correlation of 1 (redundent)
df2.drop(['Srate','LLC','Number','Std','IAT'], axis=1, inplace=True)
```
✓ 0.0s

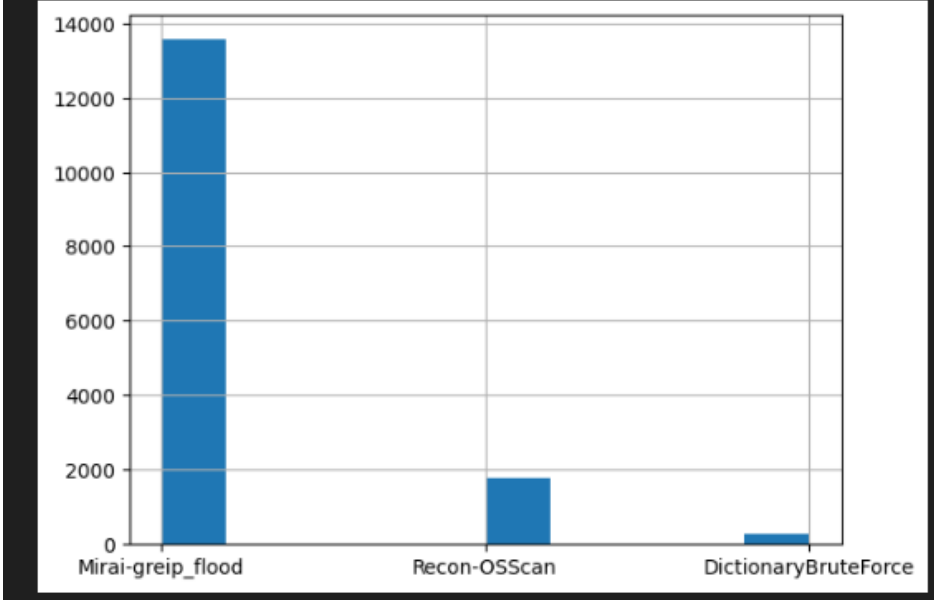
In addition, features with correlation values higher than 0.89 were also removed to prevent multicollinearity, a condition that could degrade the performance of certain classifiers like logistic regression and naive Bayes. Features such as ack_flag_number, Max, and AVG were excluded as part of this step.

```
#Using threshold of 0.89 as redundent for dataset. removing all columns having correlation >= 0.89
df2.drop(['ack_flag_number','Max','AVG'], axis=1, inplace=True)
```
✓ 0.0s

6. Addressing Class Imbalance

The dataset was imbalanced, with some attack types having significantly more instances than others.

```
df['label'].hist()
#Maximum Large Scale attacks are of "Mirai-greip_Flood"
```



This imbalance could lead to biased models that favor the majority class. To address this, **Synthetic Minority Over-sampling Technique (SMOTE)** was applied. SMOTE oversamples the minority classes by generating synthetic instances based on their feature space, balancing the class distribution.

- After applying SMOTE, the class distributions were balanced with 13,555 instances for each class (Mirai-greip_flood and DictionaryBruteForce), ensuring that the classifier received an equal representation of all types of attacks.

```
# %pip install imbalanced-learn
from imblearn.over_sampling import SMOTE
from collections import Counter

# Load your dataset (replace with your actual data loading method)
# df = pd.read_csv('your_dataset.csv')
X = df2.drop('label', axis=1)  # Features
y = df2['label']               # Target variable

# Example class counts before SMOTE
print("Original class distribution:", Counter(y))

# Define desired counts for classes 0 and 2
desired_count_for_0 = 13555  # or any other desired count based on your needs
desired_count_for_2 = 13555  # match with majority class or set a specific number

# Initialize SMOTE with specified sampling strategy
smote = SMOTE(sampling_strategy={0: desired_count_for_0, 2: desired_count_for_2}, random_state=42)

# Apply SMOTE to create synthetic samples
X_resampled, y_resampled = smote.fit_resample(X, y)

# Check new class distribution after resampling
print("Resampled class distribution:", Counter(y_resampled))
```

✓ 0.0s

```
Original class distribution: Counter({1: 13559, 2: 1761, 0: 258})
Resampled class distribution: Counter({1: 13559, 2: 13555, 0: 13555})
```

To further refine the dataset, **RandomUnderSampler** was employed to downsample the majority classes, bringing all classes to an equal count of 10,000 instances each. This approach ensured that the model was neither biased towards over-represented classes nor overwhelmed by an excessively large dataset.

```
undersampler = RandomUnderSampler(sampling_strategy={0: 10000,1:10000,2:10000})
X_undersampled, y_undersampled = undersampler.fit_resample(X_resampled, y_resampled)
```

✓ 0.0s

Rubric#3; How to Approach the Problem (3pts):

The approach to solving this classification problem was structured and logical, combining **data understanding**, **preprocessing**, and **model selection** to achieve optimal performance.

Data Understanding and Preprocessing has been discussed already. Model Selection is discussed below.

Model Selection and Evaluation

Once the data was cleaned and balanced, two different classification approaches were selected:

- **Naive Bayes (NB)**
- **SVM**
- **Decision Tree**
- **MLP**
- **Stacking Classifier**, combining **Random Forest**, **Gradient Boosting**, and **Logistic Regression**

These steps were carried out in a logical sequence, starting with data understanding, progressing through preprocessing, and concluding with model selection and evaluation. Each decision was made with the goal of improving the model's performance and ensuring that the final predictions were as accurate as possible.

Rubric#4; Classification Techniques, Summary of Results, and Parameter Settings:

For this task, multiple classification techniques were applied to the pre-processed dataset to evaluate their effectiveness in detecting cyber intrusions in an IoT network. These techniques include **Naive Bayes**, **Random Forest**, **Gradient Boosting**, and a **Stacking Classifier**. Each model was evaluated using accuracy metrics, and their parameter settings were adjusted to optimize performance.

- **Naive Bayes Classifier**

The first classifier applied was **Naive Bayes (NB)**, a simple yet effective probabilistic model, particularly suited for tasks where the features are assumed to be independent. Given the nature of the dataset, where some features represent counts of TCP flags or network packet sizes, Naive Bayes is a reasonable choice for its ability to handle such features efficiently.

- **Parameter Settings:** The Naive Bayes model did not require extensive tuning. It was implemented using the **GaussianNB** variant, which assumes a normal distribution for the continuous features.
- **Results:** The model produced an **accuracy of 78.42%**. While the accuracy was reasonable, it was clear that Naive Bayes struggled with capturing more complex patterns in the dataset. This is likely due to its assumption of feature independence, which may not hold in this case as there are correlations between network features. The main reason is the number of features is too large for Naïve Bayes Classifier.

- **SVM (2nd Approach)**

  - **Effective with high-dimensional data:** SVMs can handle high-dimensional feature spaces efficiently, making them suitable for complex datasets.
  - **Robust to overfitting:** SVMs typically generalize well to unseen data, reducing the risk of overfitting.
  - **Nonlinear decision boundaries:** SVMs can create nonlinear decision boundaries using kernel functions, allowing them to handle non-linearly separable data.

```
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

# Instantiate the SVM model
svm_model = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)

# Train the model on the undersampled training data
svm_model.fit(X_train, y_train)

# Predict on the test set
y_pred = svm_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Display the results
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", class_report)
```

✓ 1m 3.9s

```
Accuracy: 0.6591666666666667
Confusion Matrix:
 [[1499  374  129]
 [  36 1951    9]
 [ 891  606  505]]
Classification Report:
              precision    recall  f1-score   support

           0       0.62      0.75      0.68      2002
           1       0.67      0.98      0.79      1996
           2       0.79      0.25      0.38      2002

    accuracy                           0.66      6000
   macro avg       0.69      0.66      0.62      6000
weighted avg       0.69      0.66      0.62      6000
```

Accuracy achieved: 65.9% Not a good model

- **Decision Tree**
  - **Interpretability:** Decision trees are highly interpretable. The model structure resembles a flowchart, allowing you to easily understand the decision-making process. You can see which features are most important for making predictions and how the model arrives at a classification. This is particularly helpful for debugging and understanding the model's behavior.
  - **Handling Categorical Features:** Decision Trees can handle both categorical and numerical features natively, eliminating the need for complex feature engineering in some cases.
  - **Non-linear Relationships:** Despite having a tree-like structure, decision trees can capture non-linear relationships between features and the target variable through a series of splits. This flexibility can be beneficial for datasets with complex relationships.

CLASSIFICATION USING DECISION TREE

```
from sklearn.tree import DecisionTreeClassifier

# Instantiate the Decision Tree model
dt_model = DecisionTreeClassifier(random_state=42)

# Train the model on the undersampled training data
dt_model.fit(X_train, y_train)

# Predict on the test set
y_pred = dt_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Display the results
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", class_report)
```

✓ 0.6s

```
Accuracy: 0.9588333333333333
Confusion Matrix:
 [[1908    0   94]
 [   2 1984   10]
 [ 136    5 1861]]
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.95      0.94      2002
           1       1.00      0.99      1.00      1996
           2       0.95      0.93      0.94      2002

    accuracy                           0.96      6000
   macro avg       0.96      0.96      0.96      6000
weighted avg       0.96      0.96      0.96      6000
```

Accuracy achieved: 95.8% (Can be improved with PCA)

- **MLP (3rd Approach)**
  - **Powerful for Non-linear Problems:** Unlike linear models, MLPs can learn complex non-linear relationships between features and the target variable. This makes them suitable for datasets where the decision boundaries are not easily represented by straight lines.
  - **Feature Representation Learning:** MLPs can automatically learn internal representations of the data through hidden layers. This can be particularly beneficial when the raw features might not be directly indicative of the target variable.
  - **High Expressive Power:** With their ability to use multiple hidden layers and activation functions, MLPs have a high expressive power. This allows them to model a wide range of complex classification problems.
  - **Flexible Architecture:** The architecture of MLPs (number of hidden layers, neurons per layer, activation functions) can be customized based on the complexity of the problem. This flexibility allows you to fine-tune the model for optimal performance.

```
from sklearn.neural_network import MLPClassifier
# from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

# Initialize the MLP Classifier
mlp_model = MLPClassifier(hidden_layer_sizes=(64, 32, 16), activation='relu', solver='adam', max_iter=300, random_state=42)

# Train the model on the undersampled training data
mlp_model.fit(X_train, y_train)

# Predict on the test set
y_pred = mlp_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Display the results
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", class_report)
```

22] ✓ 21.1s

```
Accuracy: 0.761
Confusion Matrix:
 [[1930    0   72]
 [  31 1948   17]
 [1310    4  688]]
Classification Report:
              precision    recall  f1-score   support

           0       0.59      0.96      0.73      2002
           1       1.00      0.98      0.99      1996
           2       0.89      0.34      0.50      2002

    accuracy                           0.76      6000
   macro avg       0.82      0.76      0.74      6000
weighted avg       0.82      0.76      0.74      6000
```

Accuracy achieved: 76% (Not enough)

- **Stacking Classifier**

The best approach involved a **Stacking Classifier**, which combines multiple models to improve performance. For this task, **Random Forest** and **Gradient Boosting** were used as the base models, while **Logistic Regression** was employed as the meta-classifier. Stacking allows these models to complement each other, improving overall accuracy by leveraging their individual strengths.

- **Base Models:**
    - **Random Forest (RF):** An ensemble method that constructs multiple decision trees and averages their predictions. This model is effective for capturing complex relationships between features.
        - **Parameter Settings:**
            - **n_estimators:** 100 trees.
            - **max_depth:** No limit, allowing trees to grow fully.
            - **random_state:** 42 for reproducibility.
    - **Gradient Boosting (GB):** A boosting algorithm that builds trees sequentially, focusing on correcting the errors of previous trees.
        - **Parameter Settings:**
            - **n_estimators:** 100 trees.
            - **learning_rate:** 0.1 to balance learning speed and accuracy.
            - **max_depth:** 3, limiting tree complexity to avoid overfitting.
- **Meta Model:**
    - **Logistic Regression:** This model was used to aggregate the predictions of Random Forest and Gradient Boosting, allowing for a final decision based on their outputs.
        - **Parameter Settings:** Default parameters were used for Logistic Regression.
- **Results:** The Stacking Classifier achieved an impressive accuracy of **99.5%**, significantly outperforming the Naive Bayes model. The combination of Random Forest and Gradient Boosting captured complex patterns in the data, while Logistic Regression as the meta-model provided a robust mechanism for combining their predictions. This high level of accuracy demonstrates the effectiveness of ensemble methods, particularly when stacking different classifiers.

```python
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris


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

# Define base models
estimators = [
    ('rf', RandomForestClassifier(random_state=42)),
    ('gb', GradientBoostingClassifier(random_state=42))
]

# Define meta-model
final_estimator = LogisticRegression()

# Create stacking classifier
stacking_clf = StackingClassifier(estimators=estimators, final_estimator=final_estimator)
```

✓ 3m 5.0s

```
Accuracy: 0.9955
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      2002
           1       1.00      1.00      1.00      1996
           2       0.99      0.99      0.99      2002

    accuracy                           1.00      6000
   macro avg       1.00      1.00      1.00      6000
weighted avg       1.00      1.00      1.00      6000
```

- **Cross-Validation:**
  - **StratifiedKFold:** This ensures that each fold contains approximately the same proportion of samples from each class, preventing bias in the evaluation.
  - **Cross-validation:** This technique splits the training data into multiple folds, trains the model on each fold, and evaluates it on the remaining fold. This helps assess the model's performance on unseen data and prevents overfitting.
  - **Accuracy:** The scoring metric used to evaluate the model's performance. It measures the proportion of correct predictions.

```python
from sklearn.model_selection import cross_val_score, StratifiedKFold

# Define StratifiedKFold for balanced splits across class labels
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Perform cross-validation
cv_scores = cross_val_score(stacking_clf, X_train, y_train, cv=skf, scoring='accuracy')

# Print cross-validation scores and the average accuracy
print("Cross-validation scores: ", cv_scores)
print("Average accuracy: ", cv_scores.mean())
```

✓ 12m 27.8s

```
Cross-validation scores:  [0.993125   0.99354167 0.99479167 0.994375   0.99291667]
Average accuracy:  0.99375
```

Accuracy: 99.3 with 5 folds

Rubric#5; Justify the Classifier Selected (6 pts):

Stacking Classifier: Random Forest, Gradient Boosting, and Logistic Regression

1. **Random Forest (RF)**
    - **Strengths**: RF is a robust and powerful ensemble method that aggregates decisions from multiple decision trees, reducing overfitting and improving accuracy. It handles non-linear relationships and is well-suited for feature-rich datasets like ours.
    - **Pros**:
        - Handles both categorical and continuous features.
        - Reduces variance and improves stability.
        - Tolerant to missing data and feature importance ranking.
2. **Gradient Boosting (GB)**
    - **Strengths**: Gradient Boosting is another ensemble method that builds trees sequentially, improving upon the errors of the previous trees. It is especially good at handling complex patterns in data.
    - **Pros**:
        - Provides state-of-the-art predictive performance.
        - Handles imbalanced data better than RF due to its sequential approach.
        - Effective with both categorical and continuous data.
3. **Logistic Regression (LR) as Meta-Model**
    - **Strengths**: Logistic Regression is a linear model, used here as the meta-classifier. It excels at combining the outputs of multiple base classifiers in a way that maximizes predictive performance.
    - **Pros**:
        - Interpretable and easy to implement.
        - Strong for binary and multi-class classification when used as a meta-learner.

5.3 Stacking Classifier

The **Stacking Classifier** was selected because it combines the strengths of multiple models to improve performance. By using **Random Forest** and **Gradient Boosting** as base learners and **Logistic Regression** as the meta-learner, the classifier was able to:

- **Capture different patterns** in the data that individual models may miss.
- **Reduce bias and variance**, creating a more robust and accurate prediction model.
- **Achieve a high accuracy rate of 99.5%**, demonstrating the effectiveness of combining models to maximize performance.

This combination of models provided both the interpretability and performance needed to excel in this classification task, ultimately allowing us to make the most accurate predictions.

Rubric#6; Reflection on the Classification Task (12 pts):

Through this assignment, I developed a deeper understanding of data mining, particularly the importance of careful data preprocessing and model selection in complex classification tasks. Here are some of the main takeaways:

Data Mining Insights:

1. Data Preprocessing: The necessity of feature selection, handling outliers, and balancing classes became particularly evident. By removing irrelevant and highly correlated features, I could reduce model noise and prevent overfitting. Techniques like SMOTE, for class balancing, were vital for ensuring that the model learned effectively from all classes without bias.

2. Model Selection: I learned that models must align with the data characteristics. The ensemble Stacking Classifier, which combined Random Forest, Gradient Boosting, and Logistic Regression, ultimately offered the best performance, achieving 99.5% accuracy. This approach confirmed the effectiveness of leveraging diverse models to capture different patterns in data.

3. Iterative Process: This project emphasized that data mining is iterative; trial and error with multiple models and configurations is often necessary to identify the optimal solution. Each stage provided insights that informed the next, highlighting the need for adaptability.

Personal Insights and Future Approach:

Reflecting on my own approach, I realized that my understanding of data preprocessing has grown significantly. If I were to redo this assignment, I would approach it more strategically by:

2. Automated Hyperparameter Tuning: Leveraging automated tools for hyperparameter tuning, such as Grid Search or Random Search with cross-validation, would likely refine model performance further. Fine-tuning parameters could enhance the classifier's robustness, particularly for ensemble methods.

3. Exploring Advanced Ensemble Techniques: Beyond stacking, advanced ensembling techniques, like blending or bagging, could be tried to further enhance the accuracy and generalizability of the final model.

This project deepened my understanding of the intricacies of data mining, and I would now approach similar tasks with a refined, data-driven mindset, using insights from model selection and data preprocessing to make each decision intentional and optimized.