Configuration Manual for Malware

1 System Requirements

This whole project takes into the account three important steps,

RAM: 16GB DDR2

OS: Windows 11 pro

Processor: i7 9th generation

Technology required: Python, Anaconda, Spyder, Streamlit

2 Code Execution

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import preprocessing
```

This code imports:
- `pandas` for data manipulation,
- `seaborn` and `matplotlib.pyplot` for data visualization,
- `sklearn.preprocessing` for data preprocessing.

```
random_sample_df = original_df.sample(n=200000, random_state=42)

df = random_sample_df.copy()
```

This code takes a random sample of 200,000 rows from `original_df` and stores it in `random_sample_df`. The `random_state=42` ensures reproducibility. Then, it creates a copy of this sample and assigns it to `df`.

```
df.replace('-', pd.NA, inplace=True)
print(df.head)
```

This code replaces all occurrences of the `'-'` character in `df` with `pandas`' missing value indicator (`pd.NA`). The `inplace=True` argument modifies the DataFrame in place. Finally,

`df.head` (without parentheses) is printed, but it should be `df.head()` to display the first few rows of the DataFrame.

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

This code calculates the number of missing (null) values in each column of `df` and stores the result in `null_values`. Then, it prints the count of null values for each column.

```
# Plotting null values using seaborn
plt.figure(figsize=(10, 6))
sns.barplot(x=null_values.index, y=null_values)
plt.xticks(rotation=45, ha='right')
plt.xlabel('Columns')
plt.ylabel('Number of Null Values')
plt.title('Null Values in Malware_Detect_Data.csv')
plt.tight_layout()
plt.show()
```

This code creates a bar plot using Seaborn to visualize the number of null values in each column of the DataFrame.

```
# Generate a heatmap of null values
plt.figure(figsize=(12, 8))
sns.heatmap(df.isnull(), cmap='viridis', cbar=False)
plt.xlabel('Columns')
plt.ylabel('Rows')
plt.title('Heatmap of Null Values in Malware_Detect_Data.csv')
plt.show()
```

This code generates a heatmap to visualize the distribution of null values across the DataFrame `df`.

```
selected_columns = ['duration', 'label']

selected_df = df[selected_columns]

correlation_matrix = selected_df.corr()

plt.figure(figsize=(10, 8))

sns.heatmap(correlation_matrix, annot=True, cmap='viridis', fmt=".2f",
linewidths=0.5)

plt.title('Correlation Matrix: Duration and Label')
plt.show()
```

This code does the following:
1. Selects Specific Columns: Creates a new DataFrame `selected_df` with only the 'duration' and 'label' columns from `df`.
2. Computes Correlation Matrix: Calculates the correlation matrix for these columns.
3. Plots Heatmap: Displays a heatmap of the correlation matrix with annotations, using the 'viridis' color map.
The heatmap shows how strongly 'duration' and 'label' are correlated.

```
#removing null values
df.dropna(subset=['history'], inplace=True)
```

This code removes rows from `df` where the 'history' column has null values. The `inplace=True` argument modifies the DataFrame in place.

```
label_encoder = preprocessing.LabelEncoder()
df['history']= label_encoder.fit_transform(df['history'])
df['history'].unique()
```

This code performs the following steps:

1. Creates a LabelEncoder: Initializes `LabelEncoder` from Scikit-learn's preprocessing module.
2. Encodes 'history' Column: Transforms the 'history' column's categorical values into numerical labels.
3. Displays Unique Values: Shows the unique numerical values in the 'history' column after encoding.

```
df.drop(df[df['detailed-label'] == 'C&C'].index, inplace = True)
df['detailed-label'].value_counts()
```

This code performs the following actions:

1. Drops Specific Rows: Removes rows where the 'detailed-label' column has the value 'C&C'.
2. Counts Remaining Values: Displays the count of each unique value in the 'detailed-label' column after the removal.

```
#onehot encode
onehot = pd.get_dummies(df['detailed-label'])
df = df.join(onehot)
df.head()
df.drop(['detailed-label'],axis=1,inplace=True)
df.head()
```

This code performs the following steps:

1. One-Hot Encoding: Converts the 'detailed-label' column into one-hot encoded columns, creating binary columns for each unique value in 'detailed-label'.
2. Joins Encoded Data: Adds these one-hot encoded columns to the DataFrame `df`.
3. Drops Original Column: Removes the original 'detailed-label' column from `df`.
The result is a DataFrame with the original 'detailed-label' column replaced by its one-hot encoded representation.

```
df['ts'] = pd.to_numeric(df['ts'])
df['ts'].mean()
```

This code converts the 'ts' column in `df` to numeric values and then calculates the mean of the 'ts' column.

```
df['uid']= label_encoder.fit_transform(df['uid'])
# types of timestamp
df['uid'].value_counts()
```

This code performs the following steps:

1. Encodes 'uid' Column: Transforms the 'uid' column's categorical values into numerical labels using the `LabelEncoder`.
2. Counts Unique Values: Displays the count of each unique numerical value in the 'uid' column after encoding.

```
df['id.orig_h']= label_encoder.fit_transform(df['id.orig_h'])
# types of timestamp
df['id.orig_h'].value_counts()
```

This code performs the following steps:

1. Encodes 'id.orig_h' Column: Transforms the 'id.orig_h' column's categorical values into numerical labels using `LabelEncoder`.
2. Counts Unique Values: Displays the count of each unique numerical value in the 'id.orig_h' column after encoding.

```
df['id.resp_h']= label_encoder.fit_transform(df['id.resp_h'])
df['id.resp_h'].value_counts()
```

This code performs the following steps:

1. Encodes 'id.resp_h' Column: Converts the 'id.resp_h' column's categorical values into numerical labels using `LabelEncoder`.
2. Counts Unique Values: Displays the count of each unique numerical value in the 'id.resp_h' column after encoding.

```
df['id.resp_p'].value_counts()
```
This code counts and displays the number of occurrences of each unique value in the 'id.resp_p' column of `df`.

```
df['id.resp_p']= label_encoder.fit_transform(df['id.resp_p'])
df['id.resp_p'].value_counts()
```
This code performs the following steps:

1. Encodes 'id.resp_p' Column: Converts the 'id.resp_p' column's categorical values into numerical labels using `LabelEncoder`.
2. Counts Unique Values: Displays the count of each unique numerical value in the 'id.resp_p' column after encoding.

```
#one hot encode proto
onehot = pd.get_dummies(df['proto'])
df = df.join(onehot)
df.head()
df.drop(['proto'],axis=1,inplace=True)
df.head()
```
This code performs the following steps:

1. One-Hot Encoding 'proto': Converts the 'proto' column into one-hot encoded columns.
2. Joins Encoded Data: Adds these one-hot encoded columns to the DataFrame `df`.
3. Drops Original Column: Removes the original 'proto' column from `df`.

The result is a DataFrame where the 'proto' column is replaced by its one-hot encoded representation.

```
#label encode conn_state
df['conn_state']= label_encoder.fit_transform(df['conn_state'])
df['conn_state'].value_counts()
```
This code performs the following steps:

1. Encodes 'conn_state' Column: Converts the 'conn_state' column's categorical values into numerical labels using `LabelEncoder`.
2. Counts Unique Values: Displays the count of each unique numerical value in the 'conn_state' column after encoding.

```
X = df.drop(['PartOfAHorizontalPortScan', 'n', 'tcp', 'udp', 'label' ,
'history' , 'id.resp_p'], axis=1)
y = df['label']
```

This code prepares the data for modeling:

1. Defines Features (`X`): Drops the columns `['PartOfAHorizontalPortScan', 'n', 'tcp', 'udp', 'label', 'history', 'id.resp_p']` from `df` to create the feature matrix `X`.
2. Defines Target (`y`): Assigns the 'label' column from `df` to `y`, which will be the target variable.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from imblearn.over_sampling import SMOTE, ADASYN
import matplotlib.pyplot as plt
import seaborn as sns
```

This code imports libraries for data manipulation, machine learning models, evaluation metrics, class imbalance handling, and visualization.

```
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(),
    "K-Nearest Neighbors": KNeighborsClassifier(),
    "XGBoost": XGBClassifier(use_label_encoder=False,
eval_metric='logloss')
}

def plot_confusion_matrix(cm, model_name, balancing_technique):
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=['Benign', 'Malicious'],
yticklabels=['Benign', 'Malicious'])
    plt.title(f'Confusion Matrix for {model_name} with
{balancing_technique}')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

results = {
    'Technique': [],
    'Model': [],
    'Accuracy': [],
    'Precision (Benign)': [],
    'Precision (Malicious)': [],
    'Recall (Benign)': [],
    'Recall (Malicious)': [],
    'F1 Score (Benign)': [],
    'F1 Score (Malicious)': []
}

techniques = {
    'Stratified Split': None,
    'SMOTE': SMOTE(random_state=42),
    'ADASYN': ADASYN(random_state=42)
}

for technique_name, technique in techniques.items():
```

```
    # Scale the features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    if technique:
        X_resampled, y_resampled = technique.fit_resample(X_scaled, y)
    else:
        X_resampled, y_resampled = X_scaled, y

    X_train, X_test, y_train, y_test = train_test_split(X_resampled,
y_resampled, test_size=0.2, random_state=42, stratify=y_resampled)

    for name, model in models.items():
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        cm = confusion_matrix(y_test, y_pred)

        report = classification_report(y_test, y_pred, output_dict=True)

        results['Technique'].append(technique_name)
        results['Model'].append(name)
        results['Accuracy'].append(accuracy_score(y_test, y_pred))
        results['Precision (Benign)'].append(report['0']['precision'])
        results['Precision (Malicious)'].append(report['1']['precision'])
        results['Recall (Benign)'].append(report['0']['recall'])
        results['Recall (Malicious)'].append(report['1']['recall'])
        results['F1 Score (Benign)'].append(report['0']['f1-score'])
        results['F1 Score (Malicious)'].append(report['1']['f1-score'])

        # Plot the confusion matrix
        plot_confusion_matrix(cm, name, technique_name)

        print(f"Technique: {technique_name}, Model: {name}")
        print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
        print(f"Classification Report:\n{classification_report(y_test,
y_pred)}")
        print(f"Confusion Matrix:\n{cm}")
        print("-" * 50)

results_df = pd.DataFrame(results)
```

```
results_df.to_csv('model_evaluation_results.csv', index=False)
print("Evaluation results saved to 'model_evaluation_results.csv'")
```

The code evaluates different machine learning models using various balancing techniques, calculates performance metrics, plots confusion matrices, and saves the results to a CSV file.

```
pivot_df = results_df.pivot_table(index=['Model', 'Technique'],
                                  values=['Accuracy', 'Precision (Benign)'
,'Precision (Malicious)',
                                          'Recall (Benign)', 'Recall
(Malicious)',
                                          'F1 Score (Benign)', 'F1 Score
(Malicious)'])

models = results_df['Model'].unique()
for model in models:
    model_df = pivot_df.xs(model, level='Model')
    model_df = model_df.round(3)
    model_df.to_csv(f'{model.replace(" ", "_")}_metrics.csv')
    print(f"Results for {model} saved to '{model.replace(' ',
'_')}_metrics.csv'")
```

This code does the following:

1. Pivot Table: Creates a pivot table from `results_df` to summarize metrics by model and balancing technique.
2. Save Metrics: Iterates over each model, rounds the metrics to three decimal places, and saves them to a CSV file named after the model.

Each model's performance metrics are saved in a separate file for easier analysis.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout
from tensorflow.keras.utils import to_categorical
from imblearn.over_sampling import SMOTE, ADASYN

# Define a function to prepare and evaluate models
def prepare_and_evaluate(X_train, X_test, y_train, y_test,
technique_name):
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    label_encoder = LabelEncoder()
    y_train_encoded = label_encoder.fit_transform(y_train)
    y_test_encoded = label_encoder.transform(y_test)
    y_train_categorical = to_categorical(y_train_encoded)
    y_test_categorical = to_categorical(y_test_encoded)

    # ANN Model
    model_ann = Sequential([
        Dense(64, activation='relu',
input_shape=(X_train_scaled.shape[1],)),
        Dense(32, activation='relu'),
        Dense(2, activation='softmax')
    ])
    model_ann.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
    model_ann.fit(X_train_scaled, y_train_categorical, epochs=10,
batch_size=32, validation_split=0.2, verbose=0)

    loss_ann, accuracy_ann = model_ann.evaluate(X_test_scaled,
y_test_categorical, verbose=0)
    y_pred_ann = model_ann.predict(X_test_scaled)
    y_pred_ann_classes = np.argmax(y_pred_ann, axis=1)
    report_ann = classification_report(y_test_encoded, y_pred_ann_classes,
output_dict=True)

    # Plot ANN Confusion Matrix
    cm_ann = confusion_matrix(y_test_encoded, y_pred_ann_classes)
    plt.figure(figsize=(8, 6))
```

```
    sns.heatmap(cm_ann, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=label_encoder.classes_,
yticklabels=label_encoder.classes_)
    plt.title(f'Confusion Matrix for ANN with {technique_name}')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

    # LSTM Model
    X_train_lstm = X_train_scaled.reshape((X_train_scaled.shape[0], 1,
X_train_scaled.shape[1]))
    X_test_lstm = X_test_scaled.reshape((X_test_scaled.shape[0], 1,
X_test_scaled.shape[1]))

    model_lstm = Sequential([
        LSTM(64, input_shape=(X_train_lstm.shape[1],
X_train_lstm.shape[2]), return_sequences=True),
        Dropout(0.2),
        LSTM(32),
        Dense(2, activation='softmax')
    ])
    model_lstm.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
    model_lstm.fit(X_train_lstm, y_train_categorical, epochs=10,
batch_size=32, validation_split=0.2, verbose=0)

    loss_lstm, accuracy_lstm = model_lstm.evaluate(X_test_lstm,
y_test_categorical, verbose=0)
    y_pred_lstm = model_lstm.predict(X_test_lstm)
    y_pred_lstm_classes = np.argmax(y_pred_lstm, axis=1)
    report_lstm = classification_report(y_test_encoded,
y_pred_lstm_classes, output_dict=True)

    # Plot LSTM Confusion Matrix
    cm_lstm = confusion_matrix(y_test_encoded, y_pred_lstm_classes)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm_lstm, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=label_encoder.classes_,
yticklabels=label_encoder.classes_)
    plt.title(f'Confusion Matrix for LSTM with {technique_name}')
```

```
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

    return {
        'Technique': technique_name,
        'ANN Accuracy': accuracy_ann,
        'ANN Precision (Benign)': report_ann['0']['precision'],
        'ANN Precision (Malicious)': report_ann['1']['precision'],
        'ANN Recall (Benign)': report_ann['0']['recall'],
        'ANN Recall (Malicious)': report_ann['1']['recall'],
        'ANN F1 Score (Benign)': report_ann['0']['f1-score'],
        'ANN F1 Score (Malicious)': report_ann['1']['f1-score'],
        'LSTM Accuracy': accuracy_lstm,
        'LSTM Precision (Benign)': report_lstm['0']['precision'],
        'LSTM Precision (Malicious)': report_lstm['1']['precision'],
        'LSTM Recall (Benign)': report_lstm['0']['recall'],
        'LSTM Recall (Malicious)': report_lstm['1']['recall'],
        'LSTM F1 Score (Benign)': report_lstm['0']['f1-score'],
        'LSTM F1 Score (Malicious)': report_lstm['1']['f1-score']
    }

# Initialize results list
results = []

# Apply techniques and evaluate
techniques = {
    'SMOTE': SMOTE(random_state=42),
    'ADASYN': ADASYN(random_state=42),
    'Stratified Split': None
}

for name, technique in techniques.items():
    if technique:
        # Apply balancing technique
        X_resampled, y_resampled = technique.fit_resample(X, y)
    else:
        # No resampling
        X_resampled, y_resampled = X, y
```

```
    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X_resampled,
y_resampled, test_size=0.2, random_state=42, stratify=y_resampled)

    # Evaluate models
    metrics = prepare_and_evaluate(X_train, X_test, y_train, y_test, name)
    results.append(metrics)

# Convert results to DataFrame
results_df = pd.DataFrame(results)
results_df.to_csv('model_performance_metrics_extended.csv', index=False)

print("Extended model performance metrics saved to
'model_performance_metrics_extended.csv'")
```

The code evaluates ANN and LSTM models with various resampling techniques, calculates performance metrics, and saves the results to a CSV file.

```
import pandas as pd



ann_columns = [col for col in results_df.columns if col.startswith('ANN')]
lstm_columns = [col for col in results_df.columns if
col.startswith('LSTM')]

results_ann = results_df[['Technique'] + ann_columns]
results_lstm = results_df[['Technique'] + lstm_columns]

results_ann.columns = [col.replace('ANN ', '') for col in
results_ann.columns]
results_lstm.columns = [col.replace('LSTM ', '') for col in
results_lstm.columns]

results_ann.to_csv('ann_metrics.csv', index=False)
results_lstm.to_csv('lstm_metrics.csv', index=False)

print("ANN metrics saved to 'ann_metrics.csv'")
print("LSTM metrics saved to 'lstm_metrics.csv'")
```

This code processes and saves metrics for ANN and LSTM models separately:

1. Filter Columns: Identifies columns related to ANN and LSTM metrics.
2. Create Separate DataFrames: Extracts relevant columns for ANN and LSTM metrics into separate DataFrames.
3. Rename Columns: Removes 'ANN ' and 'LSTM ' prefixes from column names.
4. Save to CSV: Writes the ANN and LSTM metrics to separate CSV files.

The results are saved as `ann_metrics.csv` and `lstm_metrics.csv`

```
import pandas as pd

# Define the file paths and model names
files_and_names = [
    ('/content/Random_Forest_metrics.csv', 'Random Forest'),
    ('/content/Logistic_Regression_metrics.csv', 'Logistic Regression'),
    ('/content/K-Nearest_Neighbors_metrics.csv', 'K-Nearest Neighbors'),
    ('/content/XGBoost_metrics.csv', 'XGBoost'),
    ('/content/lstm_metrics.csv', 'LSTM'),
    ('/content/ann_metrics.csv', 'ANN')
]

# Function to read each file and add a 'Model' column
def read_and_label(file_path, model_name):
    df = pd.read_csv(file_path)
    df['Model'] = model_name
    return df

# Read all files and label them
all_dfs = [read_and_label(file, model) for file, model in files_and_names]
combined_df = pd.concat(all_dfs, ignore_index=True)
```

This code combines multiple CSV files into a single DataFrame:

1. Define File Paths and Model Names: Specifies paths and names for each model's metrics file.

2. Read and Label Function: Reads each CSV file into a DataFrame and adds a column for the model name.

3. Read and Combine: Reads each file, labels it with the model name, and combines all DataFrames into one `combined_df`.
The result is a unified DataFrame with metrics from different models, labeled appropriately.

```
import pandas as pd

def combine_metrics(df):
    df['Precision'] = (df['Precision (Benign)'] + df['Precision
(Malicious)']) / 2
    df['Recall'] = (df['Recall (Benign)'] + df['Recall (Malicious)']) / 2
    df['F1 Score'] = (df['F1 Score (Benign)'] + df['F1 Score
(Malicious)']) / 2

    # Drop individual Benign and Malicious columns
    df_combined = df.drop(columns=[
        'Precision (Benign)', 'Precision (Malicious)',
        'Recall (Benign)', 'Recall (Malicious)',
        'F1 Score (Benign)', 'F1 Score (Malicious)'
    ])

    return df_combined

# Combine metrics for each model
combined_metrics = combine_metrics(average_metrics)

print(combined_metrics)
```

This code defines a function and applies it to simplify performance metrics:

1. Define `combine_metrics` Function:
   - Calculate Averages: Computes the average `Precision`, `Recall`, and `F1 Score` across 'Benign' and 'Malicious' categories.
   - Drop Columns: Removes the individual 'Benign' and 'Malicious' columns from the DataFrame.

2. Apply Function: Applies this function to the DataFrame `average_metrics` to produce `combined_metrics`.

3. Print Result: Displays the simplified DataFrame `combined_metrics`.

This produces a DataFrame with averaged performance metrics for each model.

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

def plot_metric(df, metric_name):
    plt.figure(figsize=(12, 6))
    ax = sns.barplot(x='Model', y=metric_name, data=df, palette='viridis')
    plt.title(f'{metric_name} Comparison')
    plt.xlabel('Model')
    plt.ylabel(metric_name)
    plt.xticks(rotation=45)
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    # Add values on top of bars
    for p in ax.patches:
        height = p.get_height()
        ax.annotate(f'{height:.3f}',
                    (p.get_x() + p.get_width() / 2., height),
                    ha='center', va='center',
                    xytext=(0, 5),  # 5 points vertical offset
                    textcoords='offset points')

    plt.show()

# Plot each metric with values
plot_metric(combined_metrics, 'Accuracy')
plot_metric(combined_metrics, 'Precision')
plot_metric(combined_metrics, 'Recall')
plot_metric(combined_metrics, 'F1 Score')
```

This code creates and displays bar plots for different performance metrics:

1. Define `plot_metric` Function:
   - Plotting: Creates a bar plot using Seaborn with models on the x-axis and the specified metric (`metric_name`) on the y-axis.
   - Customize Plot: Sets the plot title, labels, and rotation for x-axis ticks. Adds grid lines and value annotations on top of each bar.

2. Plot Metrics:
   - `plot_metric` Call: Generates and displays bar plots for 'Accuracy', 'Precision', 'Recall', and 'F1 Score' using the `combined_metrics` DataFrame.

This visually compares the performance of different models based on the specified metrics.