MICROSOFT MALWARE DETECTION

CODING WEEK : REPORT

Team Members:

Anicet KOBANKA	Vincent MENDY
Doha FRIAT	Jean Bedel Bekanfa GNAMIEN
Nouhaila GARGOUZ	Nour MAZOUZ

SUPERVISOR :

Mrs Kawtar ZERHOUNI

March 18th -22nd 2024

Summary :

1 Contextualization :

1.1 Initialization :

In the realm of cybersecurity, the primary goal of this project is to develop a predictive model capable of estimating the likelihood of a Windows machine being infected by different families of malware. To embark on this predictive journey, a structured approach will be undertaken, starting with an introduction to machine learning, followed by an exploration of supervised learning categories. Subsequently, theoretical aspects including the study of concordance, and classifiers like Decision Trees, Random Forests, and XGBoost will be delved into. The process will then proceed through data understanding, preprocessing, exploratory analysis, feature engineering, culminating in the construction and evaluation of predictive models. Through this comprehensive approach, the aim is to enhance malware detection capabilities and bolster Windows machine security.

1.2 Machine learning:

What is machine learning?

Machine learning (ML) is a field of computer science that allows computers to learn without being explicitly programmed. Algorithms are trained on data, and then use that training to make predictions or decisions on new data. Machine learning is widely used in many different fields, from spam filtering to medical diagnosis.

How can we classify its approaches?

There are several ways to classify machine learning approaches, but a common breakdown is by the type of data and the kind of feedback the algorithm receives during training. Here are four main categories:

• Supervised learning: In this approach, the algorithm is trained on data that includes both the input and the desired output. For instance, an email filtering system might be trained on emails that have been labeled as spam or not spam. The algorithm learns to identify patterns in the emails that differentiate spam from legitimate emails.

• Unsupervised learning: Unsupervised learning deals with unlabeled data. The goal here is to uncover hidden patterns or structures within the data. For example, an unsupervised learning algorithm might be used to group customers into different segments based on their purchasing habits.

• Semi-supervised learning: This approach combines labeled and unlabeled data for training. It can be useful when labeled data is scarce but there is a large amount of unlabeled data available.

• Reinforcement learning: In reinforcement learning, the algorithm learns through trial and error by interacting with an environment. The algorithm receives rewards for good decisions and penalties for bad decisions. This approach is used in applications like training a self-driving car.

1.3 Feature engineering:

Feature engineering is the culinary art of transforming raw data into bite-sized pieces (features) that these models can easily digest and use for predictions. This involves selecting the most relevant data points, manipulating them to extract useful information, and even crafting entirely new features specific to the problem at hand. Think of features as the ingredients in a recipe - the quality and combination determine the success of the final dish (the model's predictions). Just as new ingredients might be needed for a unique recipe, feature engineering often involves designing and training new features tailored to the specific task. By carefully transforming raw data into these features, we build the foundation for accurate and insightful machine learning models.

2 Problem statement:

2.1 Objective:

Our primary objective is to develop machine learning models that accurately predict the likelihood of malware infections on Windows machines based on various machine properties.

2.2 Dataset Description:

The dataset utilized for this endeavor comprises telemetry data obtained from Microsoft's endpoint protection solution, Windows Defender, amalgamating heartbeat and threat reports. Each row within this dataset corresponds to a unique machine identified by a MachineIdentifier, with the ground truth label, HasDetections, indicating the presence of malware on the machine. The data collection methodology adheres to stringent business constraints, emphasizing user privacy and capturing the temporal dynamics of malware infections. Despite the inherent time-series nature of malware detection, complexities arise due to the dynamic nature of machines, such as new installations, patching, and system upgrades. Furthermore, it's crucial to note that this dataset is deliberately skewed towards malware instances rather than reflecting the general population of Microsoft customers' machines.

3 Theoretical model:

3.1 Concordance:

Concordance analysis is a statistical technique used to assess the agreement between predicted probabilities and actual outcomes in survival analysis and other domains. In the context of our project, we can use concordance to evaluate the predictive performance of our machine learning models by comparing their predicted probabilities with the ground truth labels.

3.2 Decision Tree:

A decision tree is a hierarchical tree-like structure used for classification and regression tasks. It partitions the feature space into segments based on the values of input features, making decisions at each node to maximize predictive accuracy.

3.3 Random Forest:

Random forest is an ensemble learning method that constructs multiple decision trees during training and combines their predictions through voting (classification) or averaging (regression). By aggregating the predictions of individual trees, random forest models tend to generalize well and exhibit robust performance.

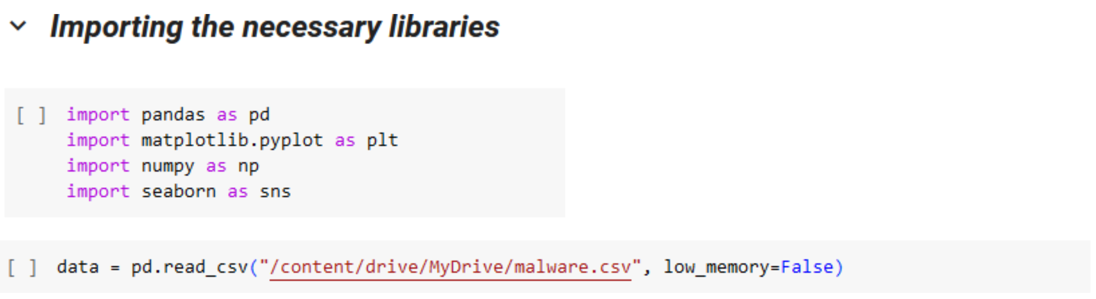
3.4 XGBoost:

XGBoost (Extreme Gradient Boosting) is a scalable and efficient implementation of gradient boosting, a machine learning technique that builds an ensemble of weak learners (typically decision trees) sequentially to minimize a predefined loss function. XGBoost is known for its high performance and flexibility, making it a popular choice for various classification and regression tasks.

4 Implemented solution:

4.1 Data exploration:

The first crucial step in any data analysis project is the process of reading the data. For this project, we utilized Google Colab as our analysis environment, a cloud-based platform that provides easy access to powerful computing resources and facilitates collaborative work. We accessed our dataset, named "malware," which was stored in a CSV file format.


Importing the necessary libraries

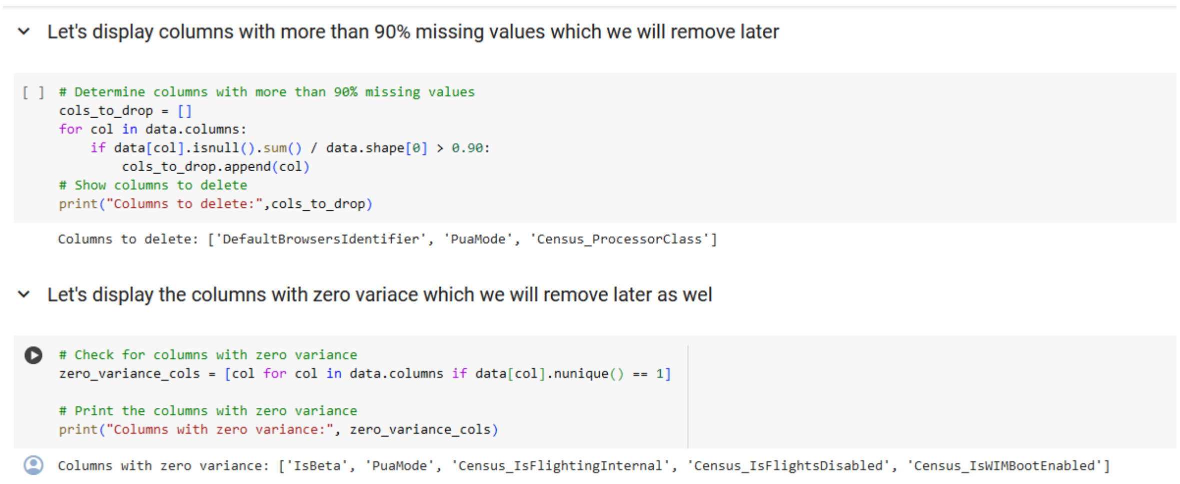
```
[ ] import pandas as pd
    import matplotlib.pyplot as plt
    import numpy as np
    import seaborn as sns
```

```
[ ] data = pd.read_csv("/content/drive/MyDrive/malware.csv", low_memory=False)
```

We begin by importing the necessary libraries for data manipulation(pandas), visualization (matplotlib and seaborn), and numerical computations(numpy).

We add low_memory=False while reading the data in order to recover all the lines of the file instead of truncating them to save memory space.

After this initial data reading step, upon closer examination, we observed several patterns that indicated the need for data preprocessing. Notably, we identified certain columns with high percentages of missing values, such as PuaMode and Census_ProcessorClass, exceeding 99%. Given the extent of missing data, these columns are unlikely to provide meaningful insights and should be dropped from the dataset. Additionally, columns Census_IsWIMBootEnabled showed no variance, providing no discriminatory power, and have therefore to be eliminated.

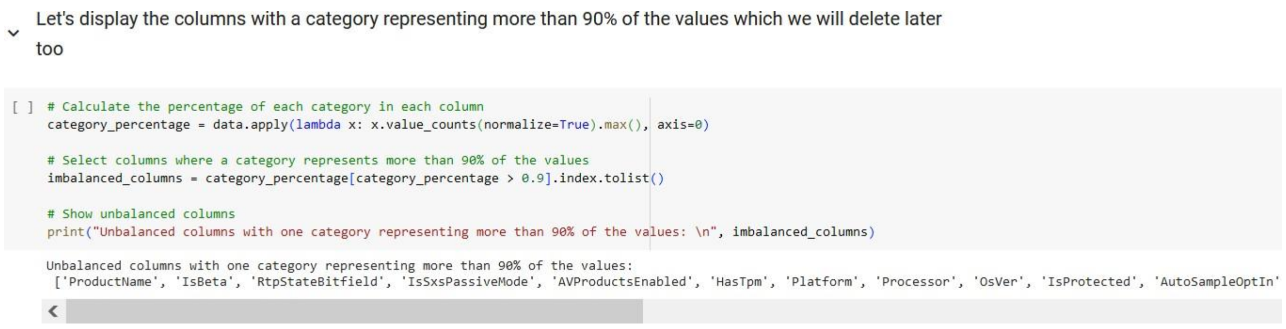

Let's display columns with more than 90% missing values which we will remove later

```
[ ] # Determine columns with more than 90% missing values
    cols_to_drop = []
    for col in data.columns:
        if data[col].isnull().sum() / data.shape[0] > 0.90:
            cols_to_drop.append(col)
    # Show columns to delete
    print("Columns to delete:",cols_to_drop)

    Columns to delete: ['DefaultBrowsersIdentifier', 'PuaMode', 'Census_ProcessorClass']
```

Let's display the columns with zero variace which we will remove later as wel

```
# Check for columns with zero variance
zero_variance_cols = [col for col in data.columns if data[col].nunique() == 1]

# Print the columns with zero variance
print("Columns with zero variance:", zero_variance_cols)

Columns with zero variance: ['IsBeta', 'PuaMode', 'Census_IsFlightingInternal', 'Census_IsFlightsDisabled', 'Census_IsWIMBootEnabled']
```

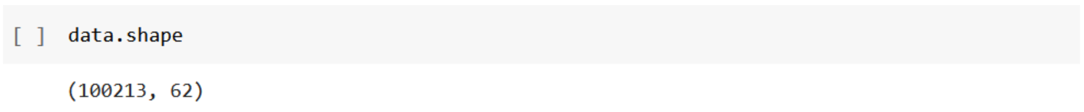
Furthermore, we noticed 27 columns where a single category encompassed over 90% of the values, indicating an imbalance that may warrant their exclusion from the dataset.


Let's display the columns with a category representing more than 90% of the values which we will delete later too

```
[ ] # Calculate the percentage of each category in each column
    category_percentage = data.apply(lambda x: x.value_counts(normalize=True).max(), axis=0)

    # Select columns where a category represents more than 90% of the values
    imbalanced_columns = category_percentage[category_percentage > 0.9].index.tolist()

    # Show unbalanced columns
    print("Unbalanced columns with one category representing more than 90% of the values: \n", imbalanced_columns)

    Unbalanced columns with one category representing more than 90% of the values:
     ['ProductName', 'IsBeta', 'RtpStateBitfield', 'IsSxsPassiveMode', 'AVProductsEnabled', 'HasTpm', 'Platform', 'Processor', 'OsVer', 'IsProtected', 'AutoSampleOptIn'
```

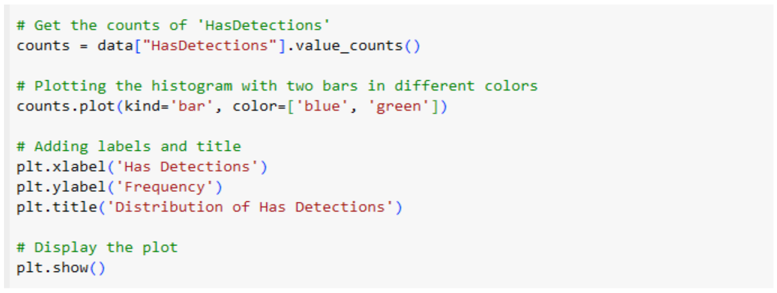
4.2 Data cleaning:

Data cleaning involves the systematic process of improving the quality and usability of a dataset. In our project, this is achieved by removing columns with a high percentage of missing values, dropping irrelevant columns, such as those related to system settings, and eliminating columns with uniform or insignificant values. After cleaning our data we had 62 columns instead of 83.

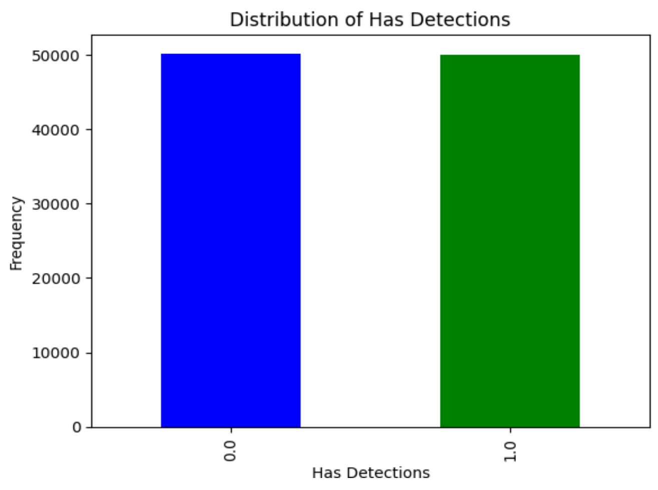


```
[ ] data.shape

    (100213, 62)
```

Before proceeding to the next step, checking if the data is balanced would be interesting.



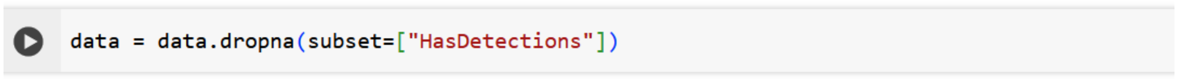
```
# Get the counts of 'HasDetections'
counts = data["HasDetections"].value_counts()

# Plotting the histogram with two bars in different colors
counts.plot(kind='bar', color=['blue', 'green'])

# Adding labels and title
plt.xlabel('Has Detections')
plt.ylabel('Frequency')
plt.title('Distribution of Has Detections')

# Display the plot
plt.show()
```



Interpretation: The number of machines affected by malware is identical to those unaffected. The data is balanced.

4.3 Feature engineering:

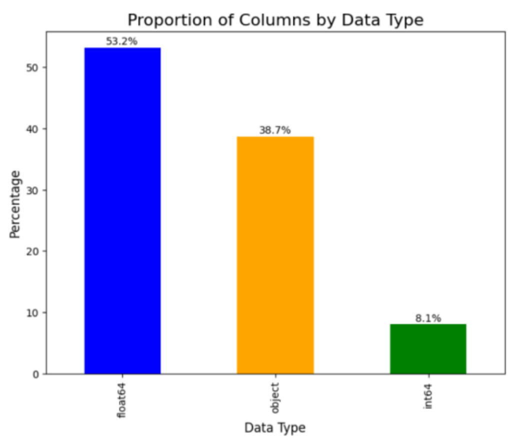
In this project, feature engineering involved replacing missing values in non-binary float and int type columns with the average of the existing values. However, before proceeding, we applied a code to remove empty lines from the HasDetections column. Indeed, it is safer to remove them than to fill them in, to avoid biasing our model.



```
data = data.dropna(subset=["HasDetections"])
```

At first, we executed the provided code located below to gain a deeper understanding of our database, aiming to discern the categories it encompasses. The outcome revealed a predominant presence of float64 values within our database, while int64 values appeared to be comparatively less prominent. This insight sheds light on the composition of our dataset, highlighting the prevalence of numerical data primarily in floating-point format, with integer values playing a relatively minor role.

```python
# Get the count of columns by data type
column_counts = data.dtypes.value_counts()

# Calculate the proportion of columns by data type
column_proportions = column_counts / len(data.columns) * 100

# Define colors for each data type
colors = ['blue', 'orange', 'green']

# Plotting the proportions of columns by data type
plt.figure(figsize=(8, 6))
column_proportions.plot(kind='bar', color=colors)
plt.title('Proportion of Columns by Data Type', fontsize=16)
plt.xlabel('Data Type', fontsize=12)
plt.ylabel('Percentage', fontsize=12)

# Adding percentages on top of each bar
for i, proportion in enumerate(column_proportions):
    plt.text(i, proportion + 0.5, f'{proportion:.1f}%', ha='center', fontsize=10)

plt.show()
```



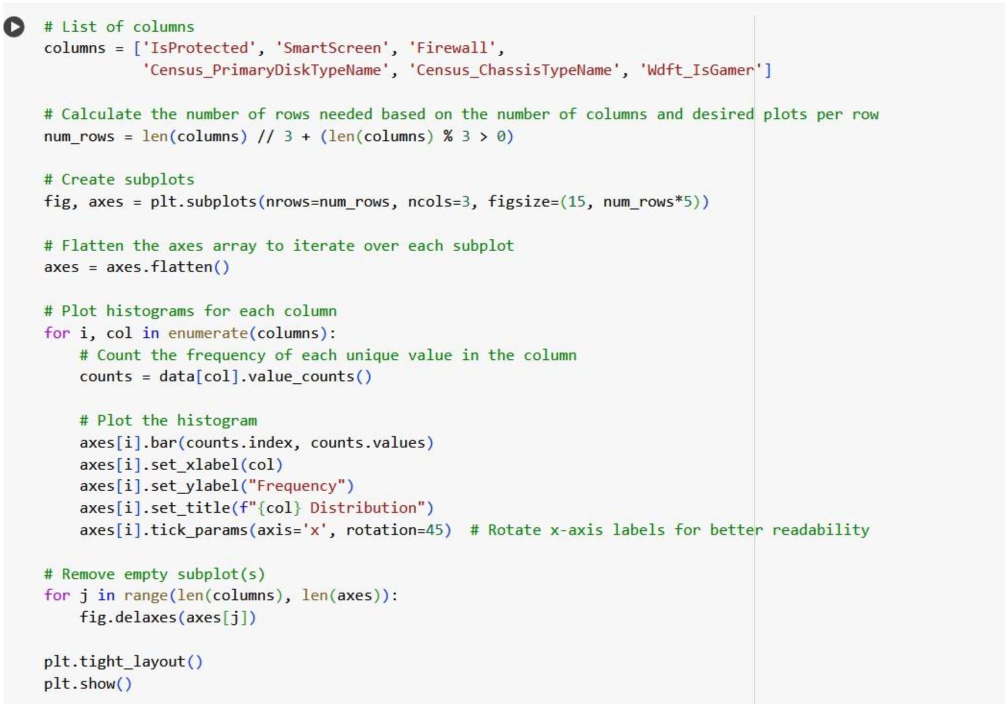
We also utilized the following code to populate the empty cells within non-binary columns, specifically those containing values of types float64 and int64, with the mean of the existing values.

```python
columns_null = [column for column in data.columns[data.isnull().any()]]
for col in columns_null:
    if data[col].dtype in ["float64", "int64"] and col not in col_binary:
        data[col] = data[col].fillna(data[col].mean())
```

For binary columns, we applied the above code to maintain the same distribution after filling the cells. Additionally, we considered using the mode, but it seemed to us that this method would have less impact on our database.

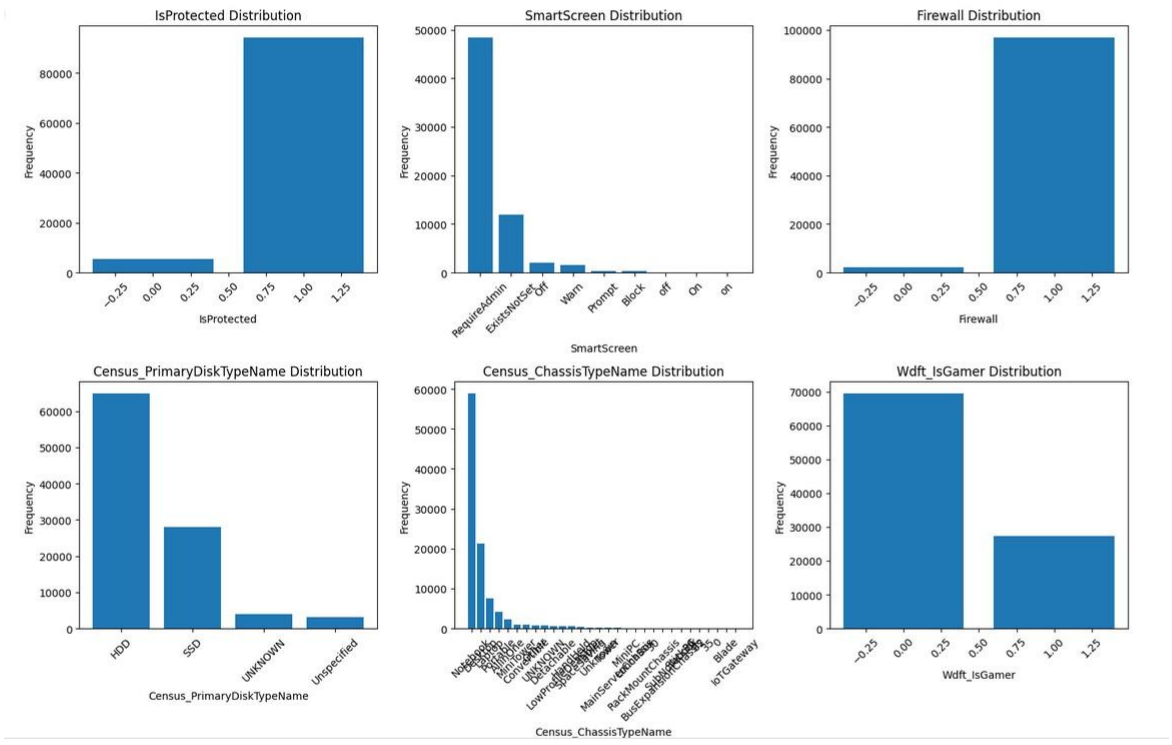
```python
# Function to fill missing values keeping the same distribution
def remplir_valeurs_manquantes(colonne):
    # Calculate frequencies of values in column
    freq_valeurs = colonne.value_counts(normalize=True)
    # Generate an array of values with the same proportions
    valeurs_remplies = np.random.choice(freq_valeurs.index, size=colonne.isnull().sum(), p=freq_valeurs.values)
    # Create a copy of the column to avoid the warnin
    colonne_copy = colonne.copy()
    # Replace missing values with generated values
    colonne_copy[colonne.isnull()] = valeurs_remplies
    return colonne_copy

# List of columns with missing values
columns_null = [column for column in data.columns[data.isnull().any()]]

# Fill in missing values in each column
for colonne in columns_null:
    data[colonne] = remplir_valeurs_manquantes(data[colonne])
```

Then, we utilized the following code to showcase the distribution of values both before and after the filling process, ensuring that we prevent the creation of a third category.

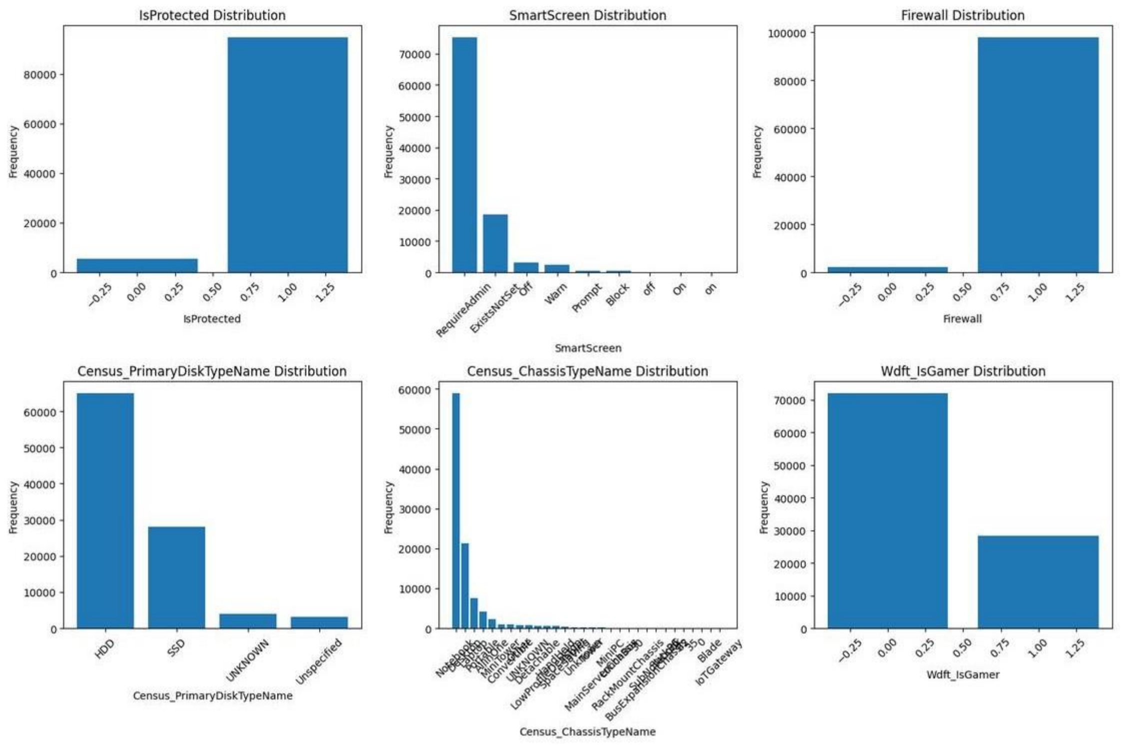


```
# List of columns
columns = ['IsProtected', 'SmartScreen', 'Firewall',
           'Census_PrimaryDiskTypeName', 'Census_ChassisTypeName', 'Wdft_IsGamer']

# Calculate the number of rows needed based on the number of columns and desired plots per row
num_rows = len(columns) // 3 + (len(columns) % 3 > 0)

# Create subplots
fig, axes = plt.subplots(nrows=num_rows, ncols=3, figsize=(15, num_rows*5))

# Flatten the axes array to iterate over each subplot
axes = axes.flatten()

# Plot histograms for each column
for i, col in enumerate(columns):
    # Count the frequency of each unique value in the column
    counts = data[col].value_counts()

    # Plot the histogram
    axes[i].bar(counts.index, counts.values)
    axes[i].set_xlabel(col)
    axes[i].set_ylabel("Frequency")
    axes[i].set_title(f"{col} Distribution")
    axes[i].tick_params(axis='x', rotation=45)  # Rotate x-axis labels for better readability

# Remove empty subplot(s)
for j in range(len(columns), len(axes)):
    fig.delaxes(axes[j])

plt.tight_layout()
plt.show()
```

Before :



After:



We observe that the distribution is indeed preserved.

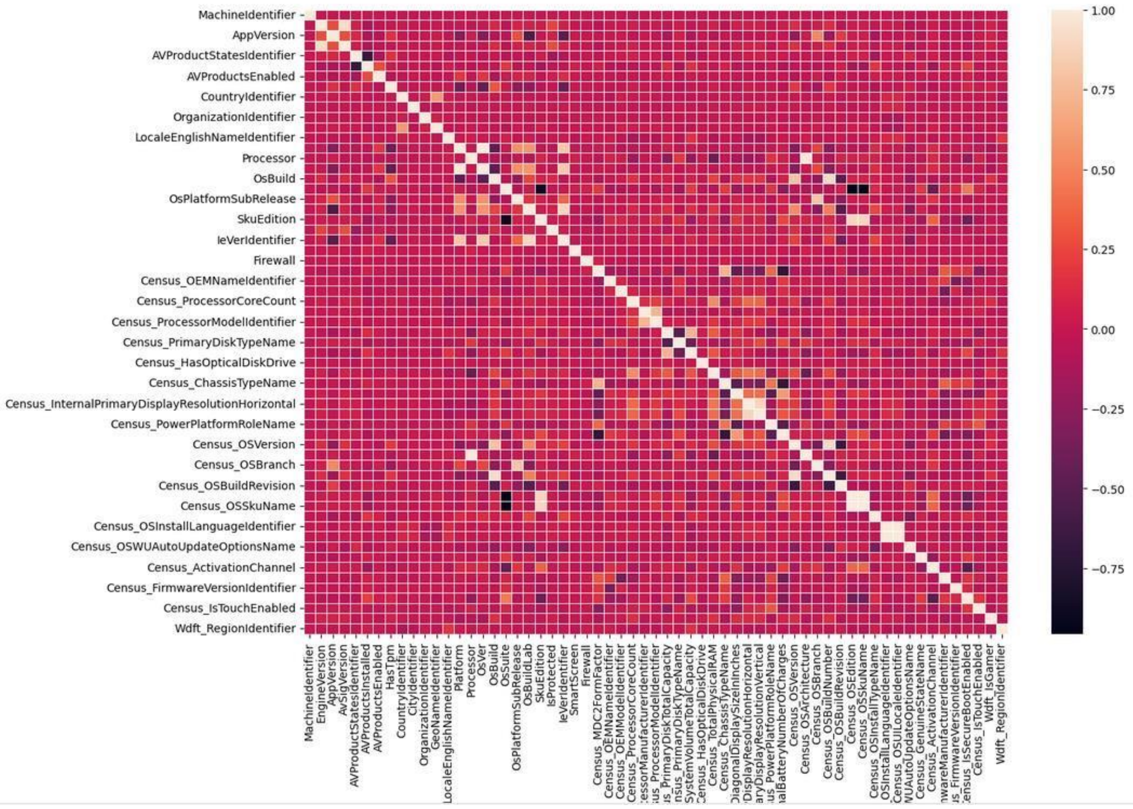
4.4 Data feature encoding:

Label encoding:

```
[ ]  from sklearn.preprocessing import LabelEncoder
```

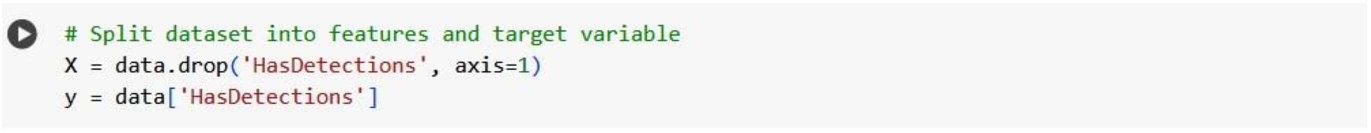
```
label_encoder = LabelEncoder()
for column in data.columns:
    data[column] = label_encoder.fit_transform(data[column])

data.head()
```

In this code, the LabelEncoder class is used to convert categorical variables into numerical values. The label_encoder object is created to perform this transformation. The code then loops through each column in the data DataFrame. For each column, the fit_transform() method of the label_encoder object is applied to convert the categorical values into numerical labels. The transformed values are then assigned back to the respective column in the data DataFrame. Finally, the data.head() function is used to display the first few rows of the transformed DataFrame.

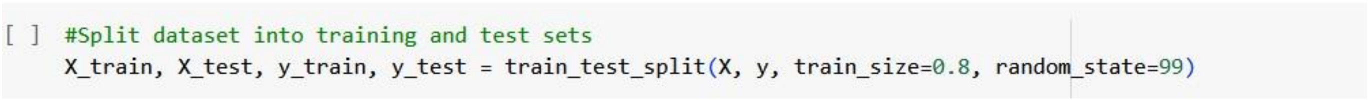


The code snippet is using the matplotlib and seaborn libraries to create a correlation heatmap of the data DataFrame. First, the code sets the size of the figure using plt.figure(figsize=(14, 10)) to create a larger plot with dimensions 14x10. Then, the sns.heatmap() function is called with the data.drop('HasDetections', axis=1).corr() parameter. This function generates a heatmap by calculating the correlation between the columns of the data DataFrame, excluding the 'HasDetections' column. The linewidths=.5 parameter sets the width of the lines between the cells in the heatmap. Finally, plt.show() is called to display the correlation heatmap.

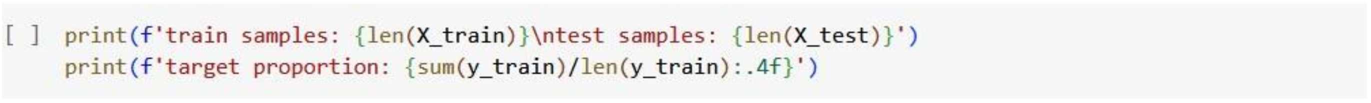
Data processing:



```
# Split dataset into features and target variable
X = data.drop('HasDetections', axis=1)
y = data['HasDetections']
```



```
[ ] #Split dataset into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, random_state=99)
```



```
[ ] print(f'train samples: {len(X_train)}\ntest samples: {len(X_test)}')
    print(f'target proportion: {sum(y_train)/len(y_train):.4f}')
```

In these lines, we are separating our dataset into two parts. X represents the input features or independent variables, and y represents the target variable or dependent variable. We drop the column 'HasDetections' from the X dataset using the drop () function. And, we are splitting the dataset into training and test sets using the train_test_split () function from the scikit-learn library. The train_size=0.8 parameter specifies that 80% of the data will be used for training, and the remaining 20% will be used for testing. The random_state=99 parameter ensures that the split is reproducible.

4.5 Concordance index:

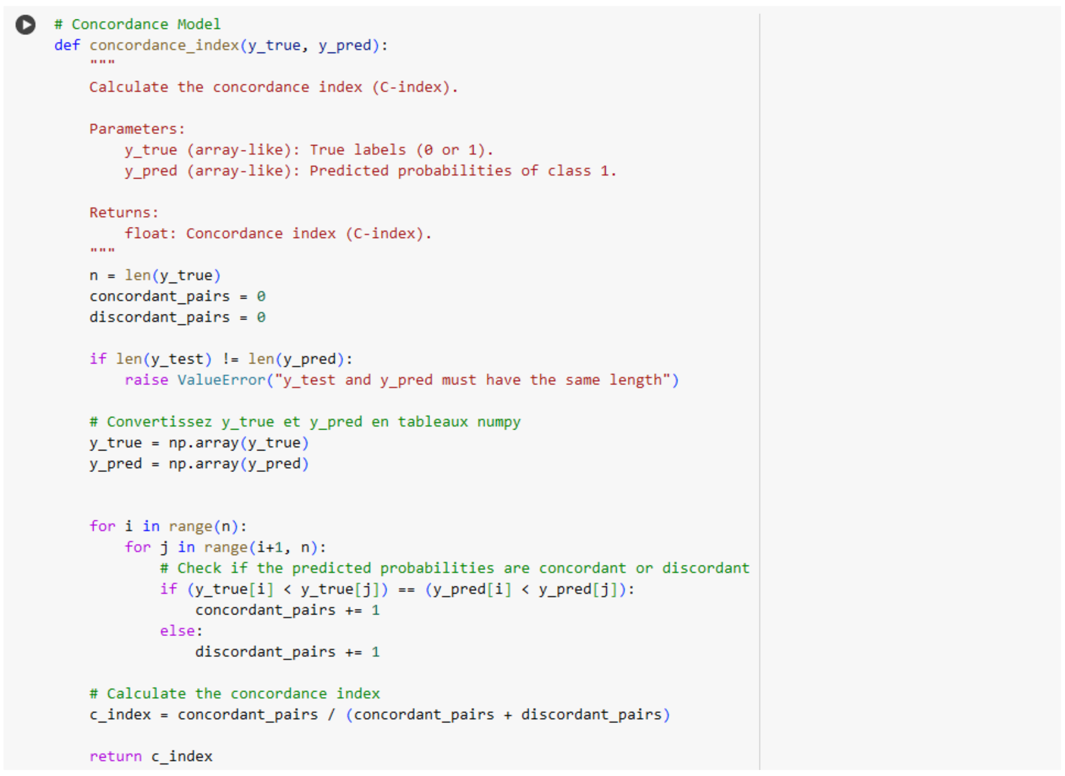
The concordance model evaluates the ability of a prediction model to correctly order samples based on their predictions.

1. Concordance: Two samples are said to be concordant if the sample with the highest probability prediction also has the highest-class label.

2. Discordance: Two samples are to be discordant if the sample with the highest probability prediction has a lower-class label than the sample with a lower probability prediction.

3. Concordance Index (C-index): This is the ratio of the number of concordant pairs to the total number of pairs of data compared. It ranges from 0 to 1, where 1 indicates perfect performance of the model in ordering samples, 0.5 indicates random performance, and 0 indicates completely incorrect performance.

In summary, the concordance model assesses a prediction model's ability to correctly rank samples based on their probability predictions, thereby providing a measure of the quality of the order predicted by the model

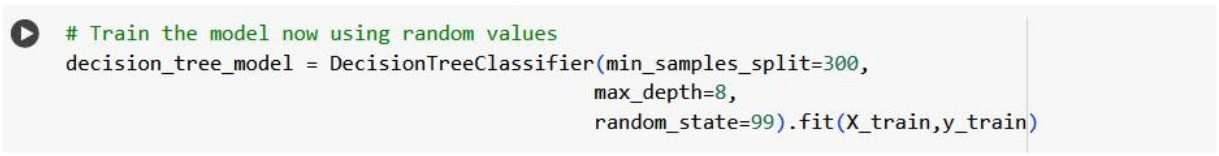


```python
# Concordance Model
def concordance_index(y_true, y_pred):
    """
    Calculate the concordance index (C-index).

    Parameters:
        y_true (array-like): True labels (0 or 1).
        y_pred (array-like): Predicted probabilities of class 1.

    Returns:
        float: Concordance index (C-index).
    """
    n = len(y_true)
    concordant_pairs = 0
    discordant_pairs = 0

    if len(y_test) != len(y_pred):
        raise ValueError("y_test and y_pred must have the same length")

    # Convertissez y_true et y_pred en tableaux numpy
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)


    for i in range(n):
        for j in range(i+1, n):
            # Check if the predicted probabilities are concordant or discordant
            if (y_true[i] < y_true[j]) == (y_pred[i] < y_pred[j]):
                concordant_pairs += 1
            else:
                discordant_pairs += 1

    # Calculate the concordance index
    c_index = concordant_pairs / (concordant_pairs + discordant_pairs)

    return c_index
```

4.6 Decision Tree model training:

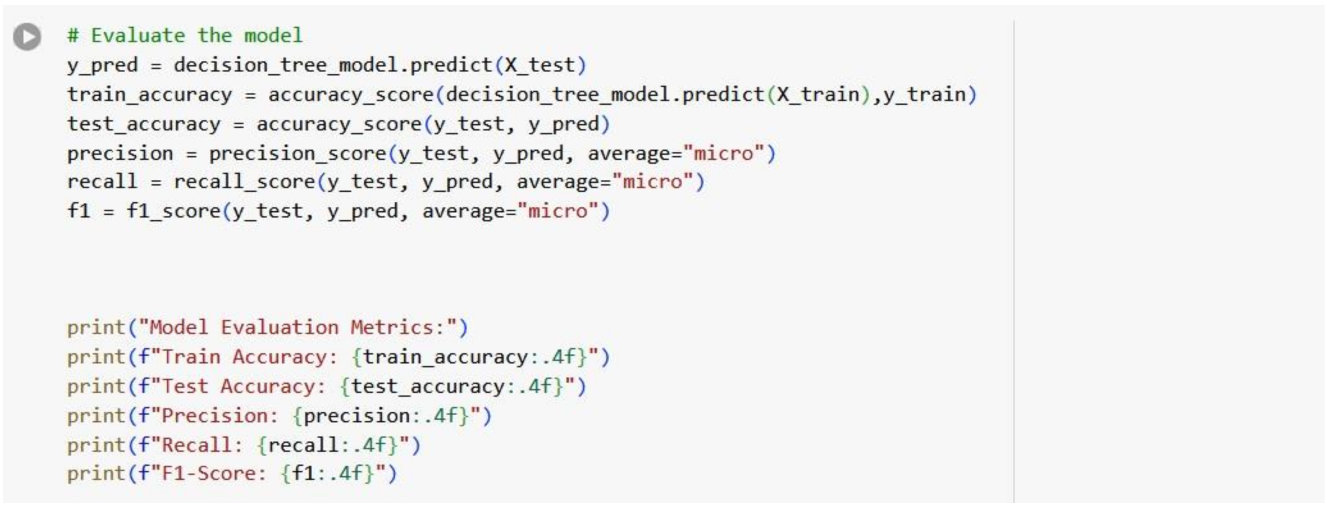
Decision tree models offer an efficient and adaptable way to classify data. Training them requires a well-defined process that unlocks their full potential. This process ensures the model can analyze and make accurate predictions on new data. Let's explore some steps involved in training a decision tree model.

First, we established the foundation by constructing the initial model with predefined hyperparameters, akin to choosing the basic recipe and ingredient quantities. Just like good starting ingredients are crucial for a delicious dish, selecting reasonable hyperparameters ensures the model has the potential to be accurate.



```python
# Train the model now using random values
decision_tree_model = DecisionTreeClassifier(min_samples_split=300,
                                             max_depth=8,
                                             random_state=99).fit(X_train,y_train)
```

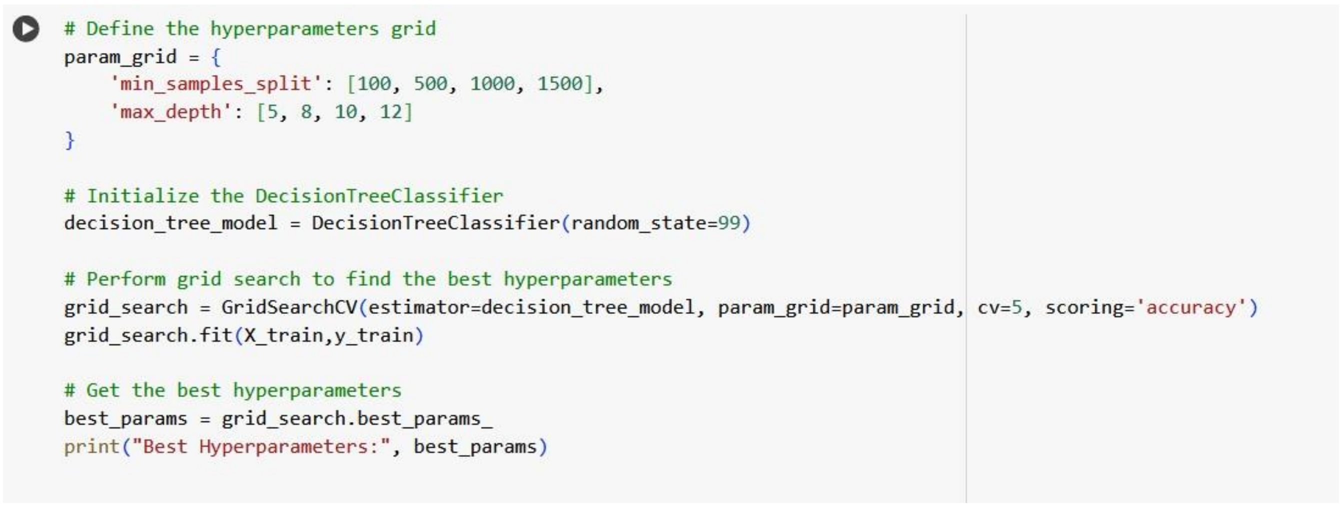
Next, we evaluated the initial model's performance. This is similar to the chef testing the first batch to identify areas for improvement. The model's performance is assessed on both familiar training data and unseen test data, providing a quantitative measure of its quality. This allows us to identify potential adjustments, analogous to tweaking spice levels, to achieve better accuracy.



```python
# Evaluate the model
y_pred = decision_tree_model.predict(X_test)
train_accuracy = accuracy_score(decision_tree_model.predict(X_train),y_train)
test_accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average="micro")
recall = recall_score(y_test, y_pred, average="micro")
f1 = f1_score(y_test, y_pred, average="micro")



print("Model Evaluation Metrics:")
print(f"Train Accuracy: {train_accuracy:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
```

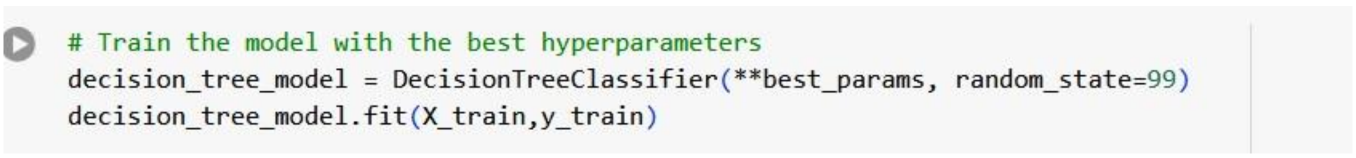
To refine the model further, we explored a range of hyperparameter options, like having a wide variety of spices at our disposal.

This code defines a search grid, essentially listing all possible combinations:



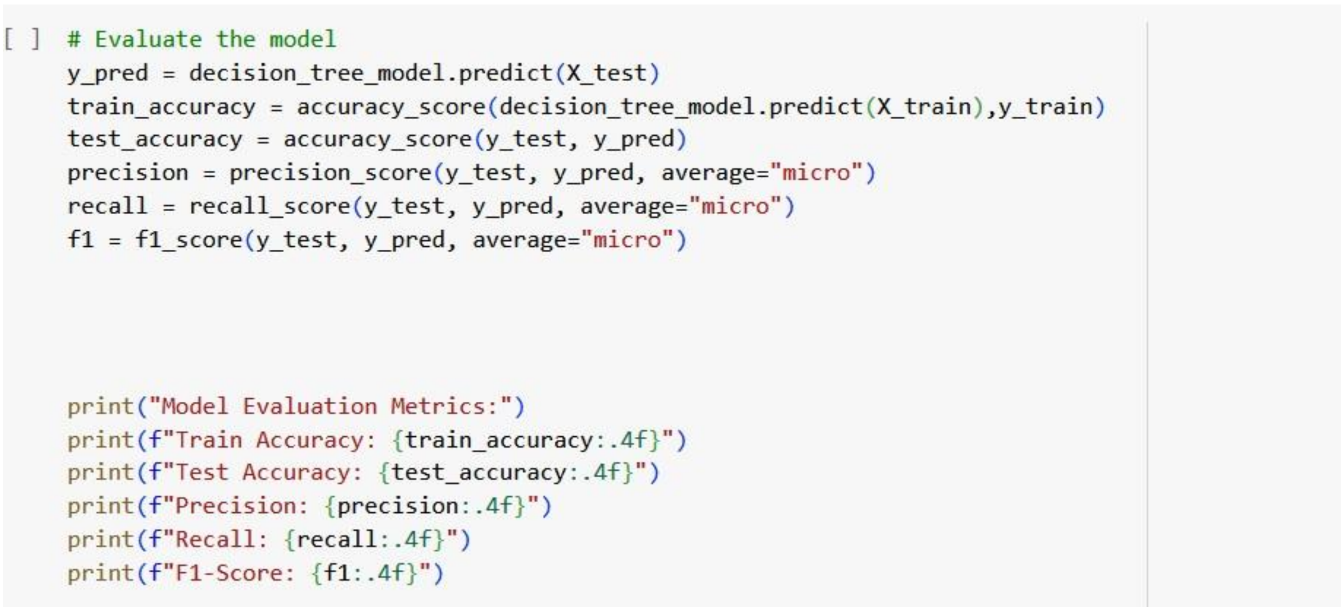
```python
# Define the hyperparameters grid
param_grid = {
    'min_samples_split': [100, 500, 1000, 1500],
    'max_depth': [5, 8, 10, 12]
}

# Initialize the DecisionTreeClassifier
decision_tree_model = DecisionTreeClassifier(random_state=99)

# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(estimator=decision_tree_model, param_grid=param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train,y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)
```

And this code acts like a skilled cook, testing these combinations (training models with different hyperparameters) to find the optimal configuration that yields the best performing model.



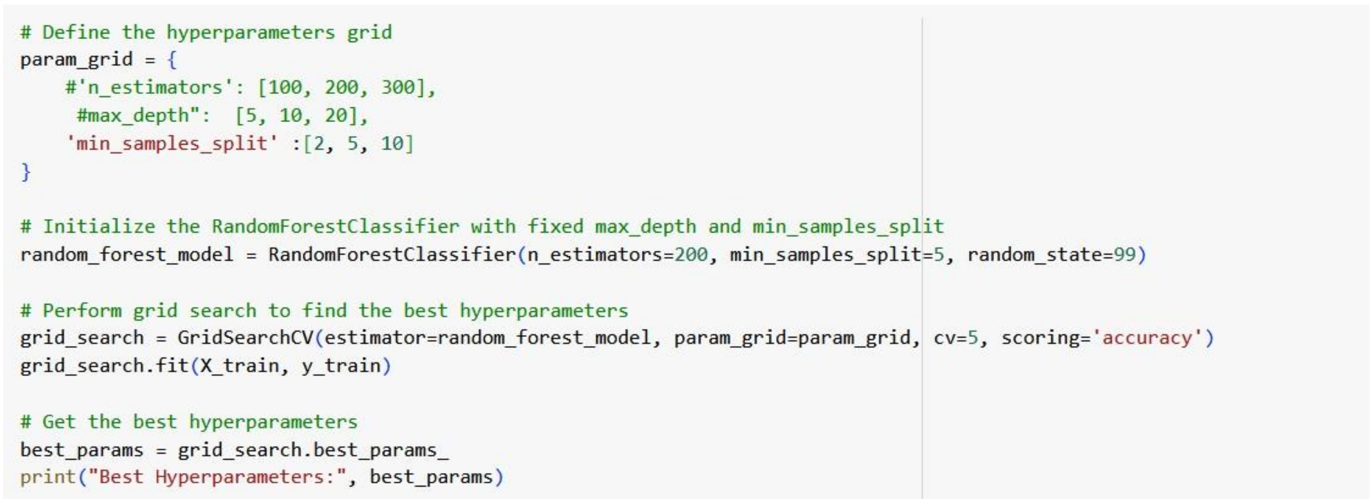
```python
# Train the model with the best hyperparameters
decision_tree_model = DecisionTreeClassifier(**best_params, random_state=99)
decision_tree_model.fit(X_train,y_train)
```

Finally, we re-evaluated the model after optimization, similar to the chef tasting the final dish. This step confirms if the adjustments led to performance improvement. By re-evaluating with the best-found hyperparameters, we ensure, like the final taste test, that the model delivers the most flavorful (accurate) outcome.



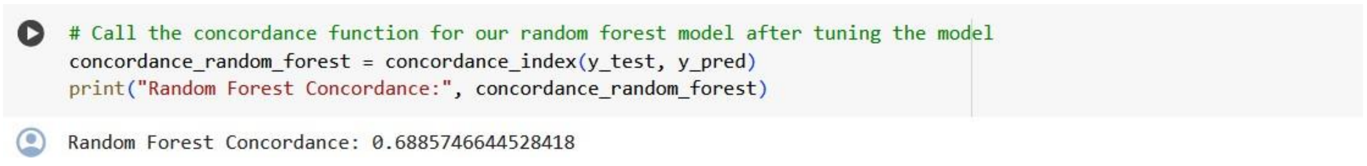
```python
[ ] # Evaluate the model
    y_pred = decision_tree_model.predict(X_test)
    train_accuracy = accuracy_score(decision_tree_model.predict(X_train),y_train)
    test_accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average="micro")
    recall = recall_score(y_test, y_pred, average="micro")
    f1 = f1_score(y_test, y_pred, average="micro")




    print("Model Evaluation Metrics:")
    print(f"Train Accuracy: {train_accuracy:.4f}")
    print(f"Test Accuracy: {test_accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")
```

4.7 Random Forest model training:



```
# Define the hyperparameters grid
param_grid = {
    #'n_estimators': [100, 200, 300],
     #max_depth":  [5, 10, 20],
    'min_samples_split' :[2, 5, 10]
}

# Initialize the RandomForestClassifier with fixed max_depth and min_samples_split
random_forest_model = RandomForestClassifier(n_estimators=200, min_samples_split=5, random_state=99)

# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(estimator=random_forest_model, param_grid=param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)
```

In these lines, we define the hyperparameters grid. Currently, we have commented out the 'n_estimators' and 'max_depth' parameters, and only included 'min_samples_split' with three possible values: 2, 5, and 10. We did uncomment the other parameters and add more values to the grid if as we needed to explore different combinations.



```
# Call the concordance function for our random forest model after tuning the model
concordance_random_forest = concordance_index(y_test, y_pred)
print("Random Forest Concordance:", concordance_random_forest)
```

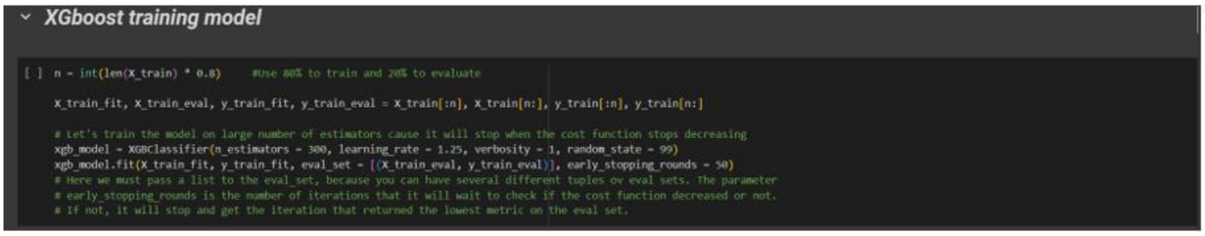
```
Random Forest Concordance: 0.6885746644528418
```

In this line, we are calling the concordance_index() function to calculate the concordance index for our random forest model. The concordance_index() function takes two arguments: y_test, which represents the true labels or target values, and y_pred, which represents the predicted labels or target values from our model.

Here, we simply print out the calculated concordance index for the random forest model. The concordance index is a measure of predictive power for survival models, where a value of 1 indicates perfect concordance and a value of 0.5 indicates random predictions. In this case, the concordance index is approximately 0.6886, indicating a moderate level of predictive power.

4.8 XGBoost Model Training:

XGBoost is a highly efficient machine learning algorithm known for its exceptional performance in structured data tasks. It employs gradient boosting, sequentially training weak learners to correct errors. XGBoost optimizes speed and accuracy through parallel computing, regularization, and advanced tree splitting.



XGboost training model

```
n = int(len(X_train) * 0.8)     #Use 80% to train and 20% to evaluate

X_train_fit, X_train_eval, y_train_fit, y_train_eval = X_train[:n], X_train[n:], y_train[:n], y_train[n:]

# Let's train the model on large number of estimators cause it will stop when the cost function stops decreasing
xgb_model = XGBClassifier(n_estimators = 300, learning_rate = 1.25, verbosity = 1, random_state = 99)
xgb_model.fit(X_train_fit, y_train_fit, eval_set = [(X_train_eval, y_train_eval)], early_stopping_rounds = 50)
# Here we must pass a list to the eval_set, because you can have several different tuples ov eval sets. The parameter
# early_stopping_rounds is the number of iterations that it will wait to check if the cost function decreased or not.
# If not, it will stop and get the iteration that returned the lowest metric on the eval set.
```

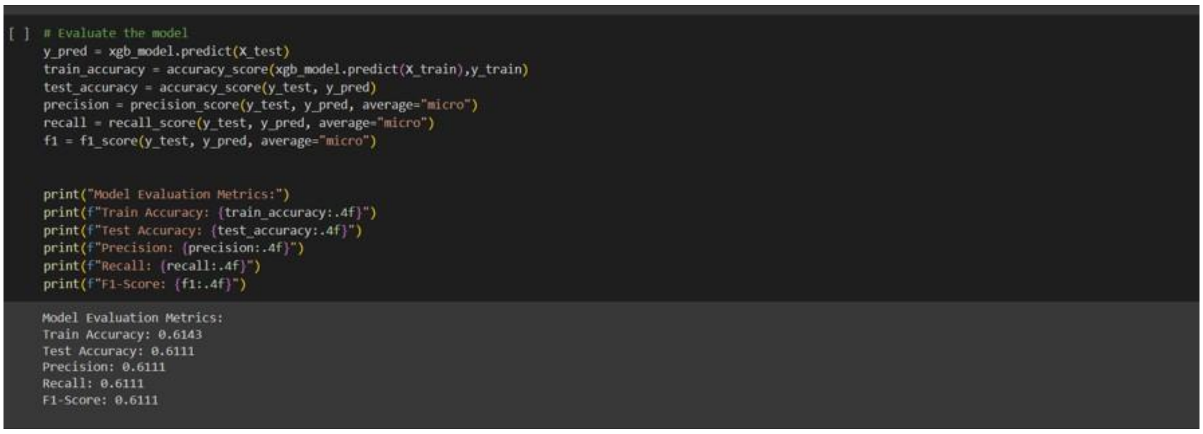
This code is for training an XGBoost Classifier model using an 80/20 train-test split and early stopping for model optimization. Here's the breakdown:

n = int(len(X_train) * 0.8): This line calculates the index to split the training data into 80% for training (X_train_fit, y_train_fit) and 20% for evaluation (X_train_eval, y_train_eval).

xgb_model = XGBClassifier(n_estimators = 300, learning_rate = 1.25, verbosity = 1, random_state = 99): Initializes an XGBoost Classifier model with specified hyperparameters, including 300 estimators (trees), a learning rate of 1.25, verbosity level set to 1 for training information display, and a random seed of 99 for reproducibility.

xgb_model.fit(X_train_fit, y_train_fit, eval_set = [(X_train_eval, y_train_eval)], early_stopping_rounds = 50): Trains the model on the training data (X_train_fit, y_train_fit), using the evaluation data specified in eval_set to monitor model performance. Early stopping is employed with a patience of 50 rounds, meaning training stops if the performance metric (e.g., log-loss) doesn't improve for 50 consecutive iterations. This helps prevent overfitting and saves computation time by stopping training when further iterations don't lead to better results.

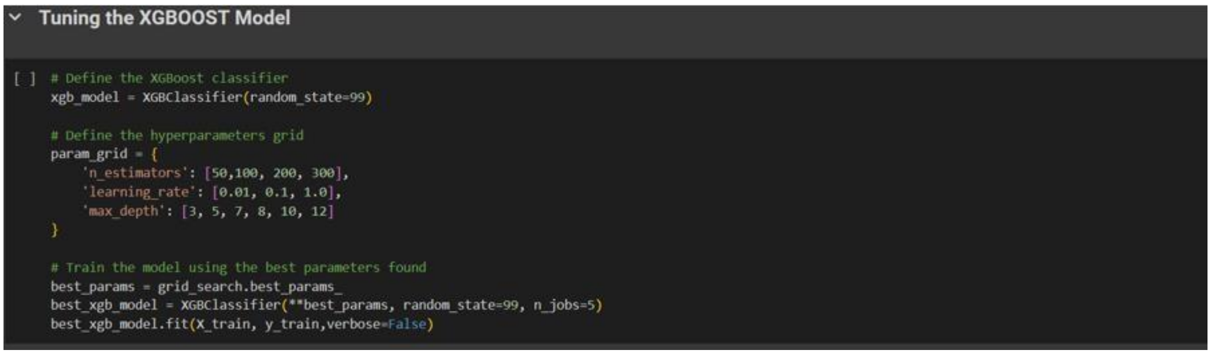
Let's evaluate the model trained with random hyperparameters value performace :



```
[ ] # Evaluate the model
    y_pred = xgb_model.predict(X_test)
    train_accuracy = accuracy_score(xgb_model.predict(X_train),y_train)
    test_accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average="micro")
    recall = recall_score(y_test, y_pred, average="micro")
    f1 = f1_score(y_test, y_pred, average="micro")


    print("Model Evaluation Metrics:")
    print(f"Train Accuracy: {train_accuracy:.4f}")
    print(f"Test Accuracy: {test_accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")
```

```
Model Evaluation Metrics:
Train Accuracy: 0.6143
Test Accuracy: 0.6111
Precision: 0.6111
Recall: 0.6111
F1-Score: 0.6111
```

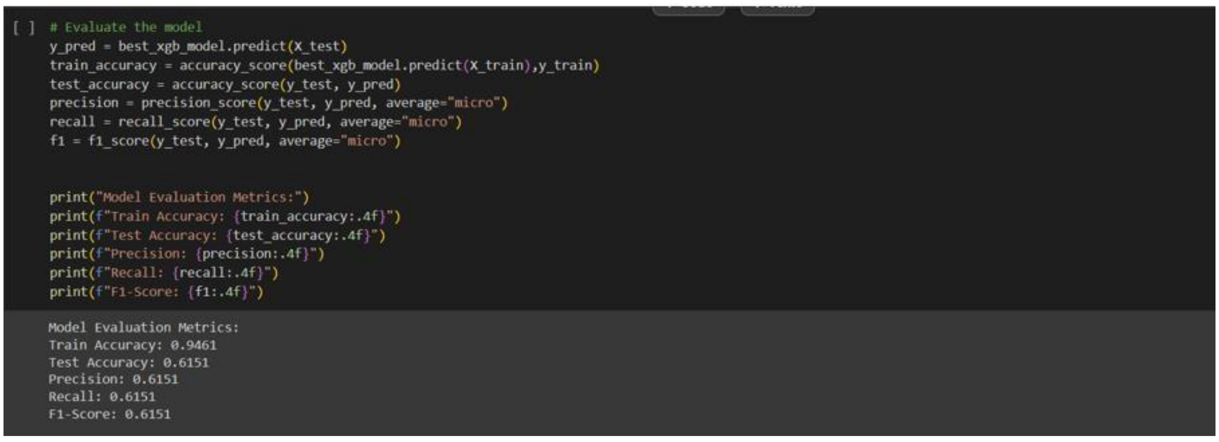
The subsequent stage involves hyperparameter tuning to identify the optimal values that significantly impact model performance.



Tuning the XGBOOST Model

```
[ ] # Define the XGBoost classifier
    xgb_model = XGBClassifier(random_state=99)

    # Define the hyperparameters grid
    param_grid = {
        'n_estimators': [50,100, 200, 300],
        'learning_rate': [0.01, 0.1, 1.0],
        'max_depth': [3, 5, 7, 8, 10, 12]
    }

    # Train the model using the best parameters found
    best_params = grid_search.best_params_
    best_xgb_model = XGBClassifier(**best_params, random_state=99, n_jobs=5)
    best_xgb_model.fit(X_train, y_train,verbose=False)
```

Now, let's assess the performance of our top-performing XGBoost model.



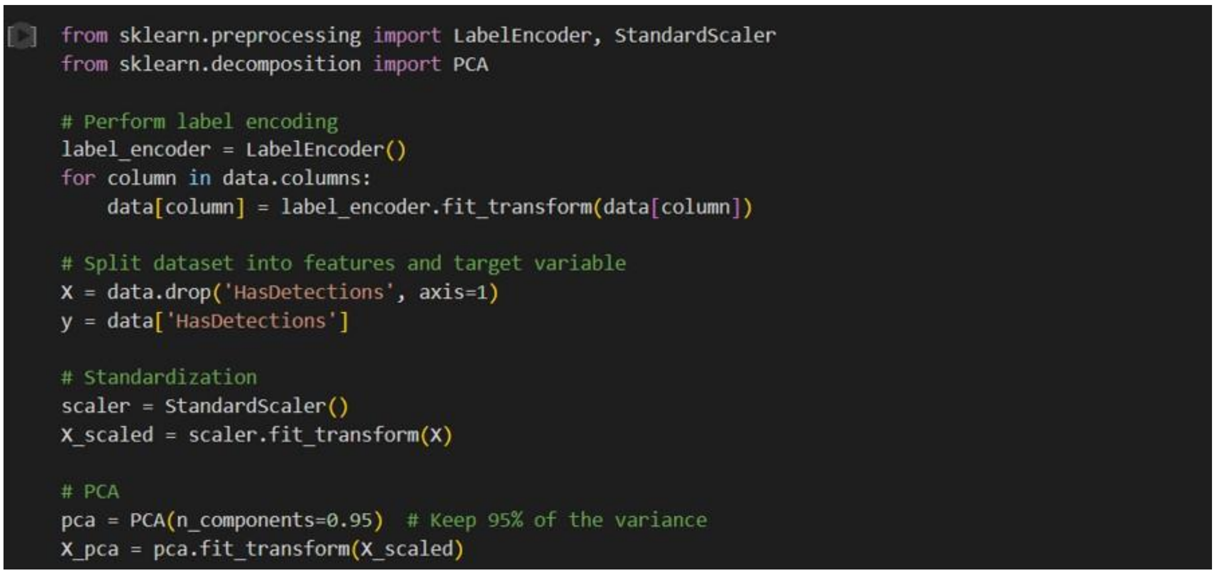
```
[ ]  # Evaluate the model
     y_pred = best_xgb_model.predict(X_test)
     train_accuracy = accuracy_score(best_xgb_model.predict(X_train),y_train)
     test_accuracy = accuracy_score(y_test, y_pred)
     precision = precision_score(y_test, y_pred, average="micro")
     recall = recall_score(y_test, y_pred, average="micro")
     f1 = f1_score(y_test, y_pred, average="micro")


     print("Model Evaluation Metrics:")
     print(f"Train Accuracy: {train_accuracy:.4f}")
     print(f"Test Accuracy: {test_accuracy:.4f}")
     print(f"Precision: {precision:.4f}")
     print(f"Recall: {recall:.4f}")
     print(f"F1-Score: {f1:.4f}")
```

```
Model Evaluation Metrics:
Train Accuracy: 0.9461
Test Accuracy: 0.6151
Precision: 0.6151
Recall: 0.6151
F1-Score: 0.6151
```

The top-performing XGBoost model showcased robust performance, boasting a training accuracy of 94.61% and a test accuracy of 61.51%. Moreover, it exhibited precision, recall, and F1-score values, all at 61.51%.

4.9 Reasons for Not Using PCA and StandardScaler Before Training Models:

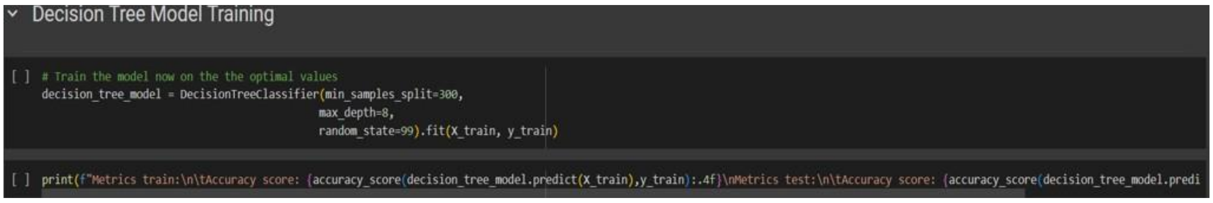
Dimensionality Reduction and Feature Scaling



```
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.decomposition import PCA

# Perform label encoding
label_encoder = LabelEncoder()
for column in data.columns:
    data[column] = label_encoder.fit_transform(data[column])

# Split dataset into features and target variable
X = data.drop('HasDetections', axis=1)
y = data['HasDetections']

# Standardization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# PCA
pca = PCA(n_components=0.95)  # Keep 95% of the variance
X_pca = pca.fit_transform(X_scaled)
```

This code performs label encoding for categorical features, standardization of features using StandardScaler, and then applies Principal Component Analysis (PCA) for dimensionality reduction while retaining 95% of the variance.

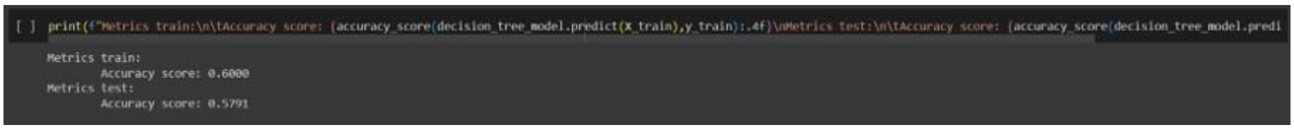
Model Training After PCA and Standardization



Decision Tree Model Training

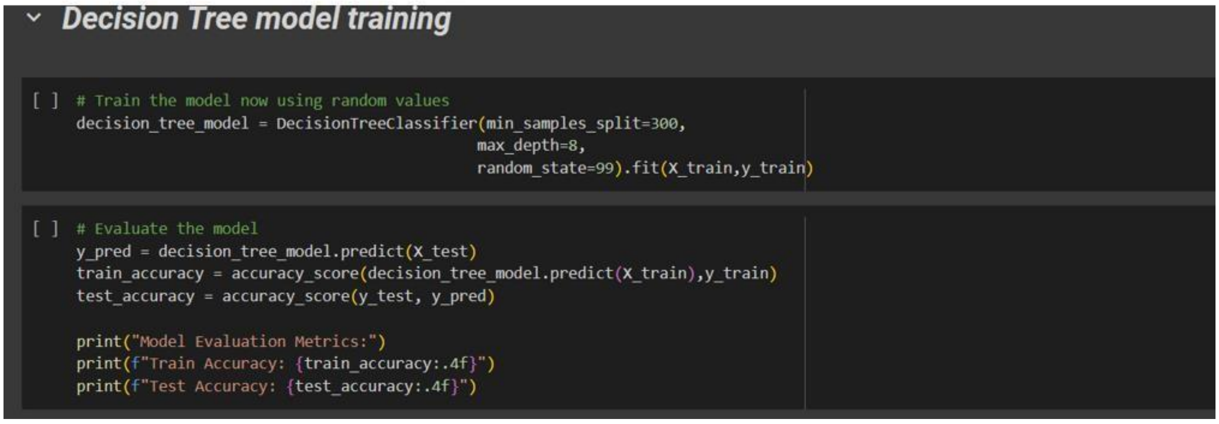
```
[ ] # Train the model now on the the optimal values
    decision_tree_model = DecisionTreeClassifier(min_samples_split=300,
                                                  max_depth=8,
                                                  random_state=99).fit(X_train, y_train)

[ ] print(f"Metrics train:\n\tAccuracy score: {accuracy_score(decision_tree_model.predict(X_train),y_train):.4f}\nMetrics test:\n\tAccuracy score: {accuracy_score(decision_tree_model.predi
```

Result:



```
[ ] print(f"Metrics train:\n\tAccuracy score: {accuracy_score(decision_tree_model.predict(X_train),y_train):.4f}\nMetrics test:\n\tAccuracy score: {accuracy_score(decision_tree_model.predi

    Metrics train:
            Accuracy score: 0.6000
    Metrics test:
            Accuracy score: 0.5791
```

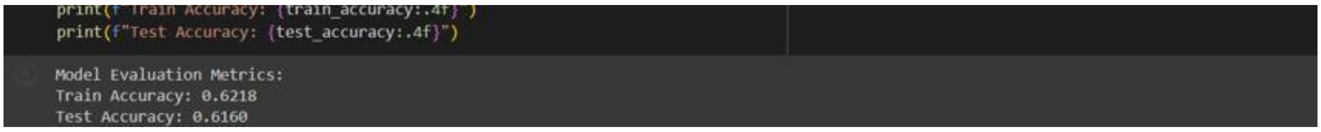
Model Training Without PCA and Standardization



Decision Tree model training

```
[ ] # Train the model now using random values
    decision_tree_model = DecisionTreeClassifier(min_samples_split=300,
                                                  max_depth=8,
                                                  random_state=99).fit(X_train,y_train)

[ ] # Evaluate the model
    y_pred = decision_tree_model.predict(X_test)
    train_accuracy = accuracy_score(decision_tree_model.predict(X_train),y_train)
    test_accuracy = accuracy_score(y_test, y_pred)

    print("Model Evaluation Metrics:")
    print(f"Train Accuracy: {train_accuracy:.4f}")
    print(f"Test Accuracy: {test_accuracy:.4f}")
```

Result:



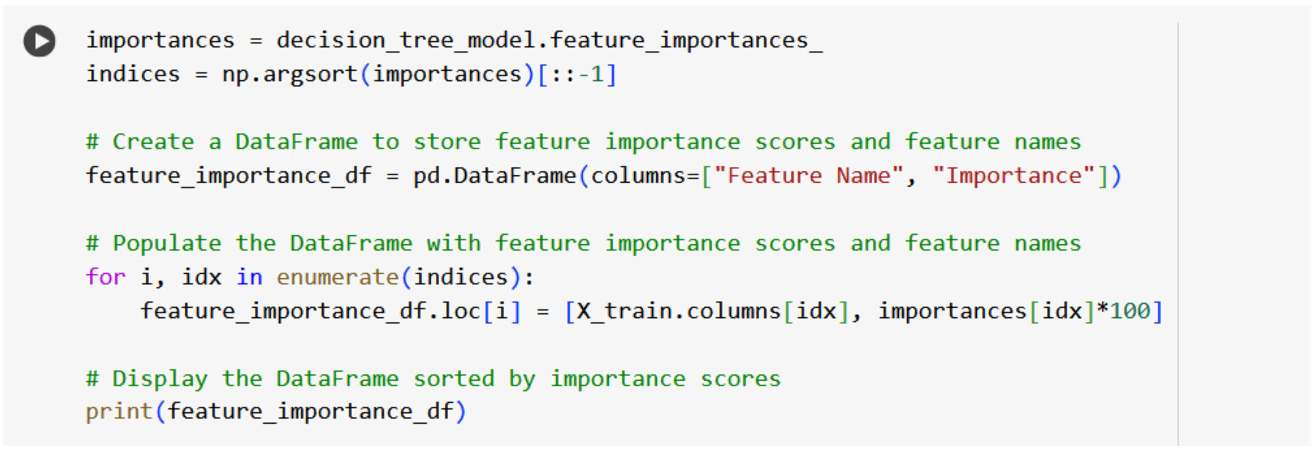
```
    print(f"Train Accuracy: {train_accuracy:.4f}")
    print(f"Test Accuracy: {test_accuracy:.4f}")

    Model Evaluation Metrics:
    Train Accuracy: 0.6218
    Test Accuracy: 0.6160
```

Conclusion:

Applying PCA and standardization didn't enhance the performance of the decision tree model. In fact, it resulted in a slight decrease in accuracy compared to the model without these techniques. Specifically, the accuracy decreased from 62.18% to 60% for the training set and from 61.60% to 57.91% for the test set. Hence, we will train our final models without PCA and standardization.

4.10 Bonus:

Decision Tree:



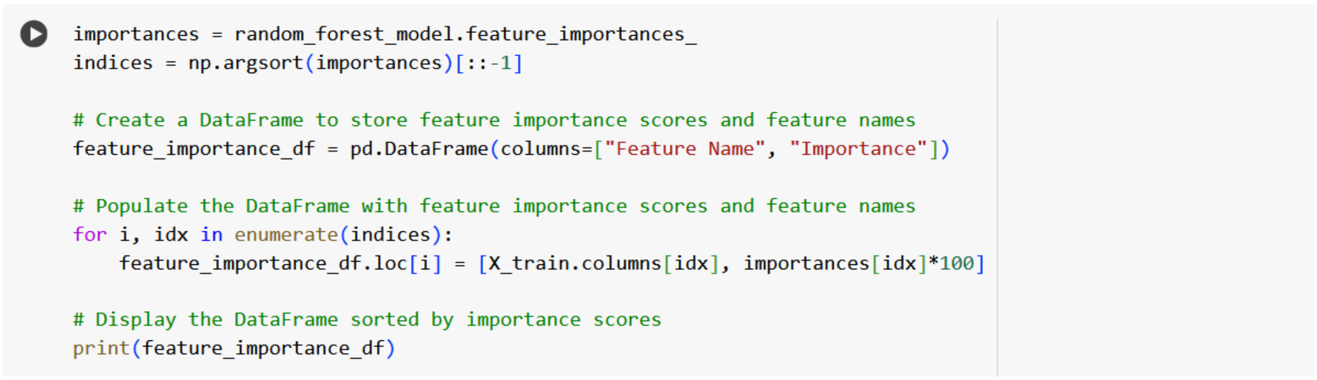
```
importances = decision_tree_model.feature_importances_
indices = np.argsort(importances)[::-1]

# Create a DataFrame to store feature importance scores and feature names
feature_importance_df = pd.DataFrame(columns=["Feature Name", "Importance"])

# Populate the DataFrame with feature importance scores and feature names
for i, idx in enumerate(indices):
    feature_importance_df.loc[i] = [X_train.columns[idx], importances[idx]*100]

# Display the DataFrame sorted by importance scores
print(feature_importance_df)
```

This code snippet calculates the importance of input features in predicting malware detections using a Decision Tree model. It computes feature importance scores and ranks them based on their contribution to the model's predictive accuracy. Analyzing these scores reveals which features are most influential in determining malware infections. This information guides feature selection and model refinement, enhancing the effectiveness of malware detection systems.

- The most important feature for the Decision Tree model is "SmartScreen" with an importance score of approximately 30.52%. This indicates that the presence or absence of SmartScreen technology on a Windows machine strongly influences the prediction of malware detection.

- Other notable features include "AVProductsInstalled" (16.55%) and "AvSigVersion" (13.22%), suggesting that the number of antivirus products installed and the antivirus signature version are also significant predictors of malware detection.

Random Forest:



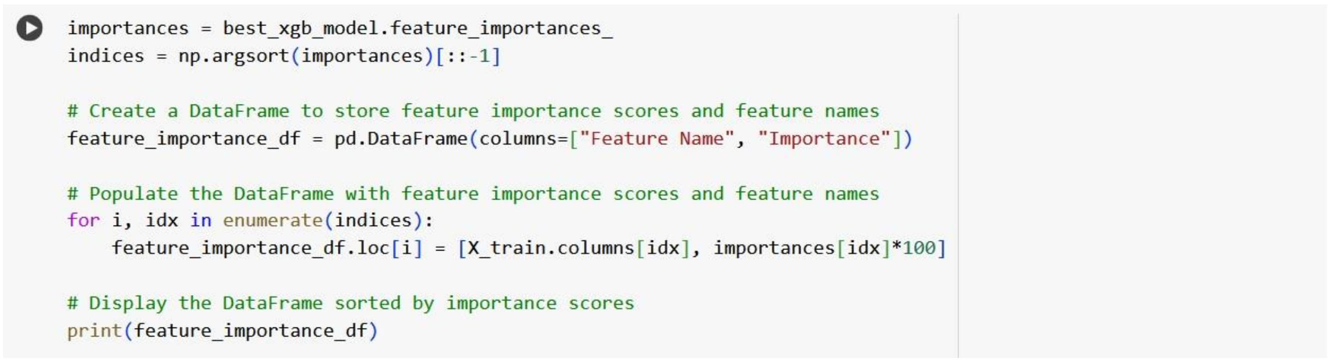
```
importances = random_forest_model.feature_importances_
indices = np.argsort(importances)[::-1]

# Create a DataFrame to store feature importance scores and feature names
feature_importance_df = pd.DataFrame(columns=["Feature Name", "Importance"])

# Populate the DataFrame with feature importance scores and feature names
for i, idx in enumerate(indices):
    feature_importance_df.loc[i] = [X_train.columns[idx], importances[idx]*100]

# Display the DataFrame sorted by importance scores
print(feature_importance_df)
```

Here, the code determines the importance of input features in predicting malware infections using a Random Forest model. By aggregating results from multiple decision trees, Random Forest improves predictive performance. Feature importance scores are calculated and ranked, highlighting the most influential variables for malware detection. This knowledge directs feature selection efforts, optimizing model performance and robustness.

- In the Random Forest model, "AvSigVersion" is identified as the most important feature, contributing approximately 5.57% to the prediction of malware detection. This highlights the importance of the antivirus signature version in determining malware infections.

- Additionally, "SmartScreen" (5.25%) and "MachineIdentifier" (4.42%) are among the top features, indicating their significant influence on the model's predictions.

XGBoost:



```
importances = best_xgb_model.feature_importances_
indices = np.argsort(importances)[::-1]

# Create a DataFrame to store feature importance scores and feature names
feature_importance_df = pd.DataFrame(columns=["Feature Name", "Importance"])

# Populate the DataFrame with feature importance scores and feature names
for i, idx in enumerate(indices):
    feature_importance_df.loc[i] = [X_train.columns[idx], importances[idx]*100]

# Display the DataFrame sorted by importance scores
print(feature_importance_df)
```

In this snippet, the importance of input features in predicting malware detections is assessed using an XGBoost model. XGBoost employs ensemble learning to combine weak learners' predictions, enhancing model accuracy. Feature importance scores generated by XGBoost identify key variables for malware detection. This insight guides feature engineering and model optimization, leading to more effective malware detection systems.

- For the XGBoost model, "SmartScreen" emerges as the most important feature, with a weight of approximately 8.29%. This underscores the crucial role of SmartScreen technology in predicting malware infections.

- Other important features include "AVProductsInstalled" (8.08%) and "HasTpm" (2.17%), suggesting that the presence of antivirus products and the Trusted Platform Module status are also relevant factors in malware detection.

5 Results and discussions (Interpretations):

- Dataset size and structure: The code reveals that the dataset contains 100,213 rows and 83 columns. This information gives us an understanding of the dataset's size and complexity.

- Missing values: The output of data.info () shows that several columns have missing values, indicated by non-null counts lower than the total number of rows. These missing values might require handling strategies, such as imputation or removal, depending on the analysis goals.

- Column data types: The code provides the data types of each column, allowing us to identify categorical variables (object type) and numerical variables (int64 and float64 types). Understanding the data types helps in selecting appropriate data preprocessing and analysis techniques.

- Potential features of interest: By examining the column names, we can identify potential features that might be relevant for the analysis. For example, columns like "AVProductsInstalled,""SmartScreen,"or"Census_InternalPrimaryDiagonalDisplaySizeInInches" could be of interest depending on the specific analysis goals.

- Exploratory data analysis: With the dataset loaded, we can perform exploratory data analysis using the imported libraries. Exploring relationships between different columns and visualizing data can help uncover patterns and insights.

- Further analysis possibilities: The code sets the foundation for further analysis. Additional code can be written to manipulate the data, perform statistical calculations, create visualizations, or develop machine learning models, depending on the specific analysis goals.

6 Conclusion and future perspectives:

Conclusion:

After hyperparameter tuning, the Decision Tree model achieved a training accuracy of 62.55% and a test accuracy of 61.09%, along with precision, recall, and F1-score values of 61.09%. The Random Forest model exhibited significantly higher accuracy, with a training accuracy of 93.71% and a test accuracy of 63.03%, maintaining consistent precision, recall, and F1-score values. Similarly, the XGBoost model demonstrated strong performance, with a training accuracy of 94.61% and a test accuracy of 61.51%, accompanied by precision, recall, and F1-score values of 61.51%.

Future Perspectives:

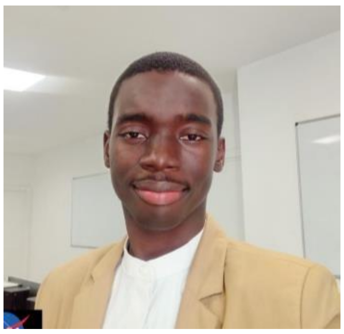
Looking ahead, further optimizations can be explored to enhance model performance. One avenue is to continue fine-tuning additional hyperparameters, including Minimum Samples Leaf, Maximum Features, Bootstrap and Subsamples, to potentially improve accuracy and other evaluation metrics. Additionally, feature selection techniques could be applied to identify the most informative features, potentially enhancing the models' predictive capabilities and reducing overfitting. Furthermore, ensemble methods, such as blending or stacking, could be investigated to leverage the strengths of multiple models and further boost overall performance. Finally, considering the entire dataset to train the model (the one used here has been truncated by the supervisor) and ongoing monitoring and validation on new data are crucial to ensure the models' robustness and effectiveness in real-world applications.

7 Team members:

Jean Bedel Bekanfa GNAMIEN

Nouhaila GARGOUZ

Doha FRIAT



Nour MAZOUZ

Anicet KOBANKA

Vincent MENDY