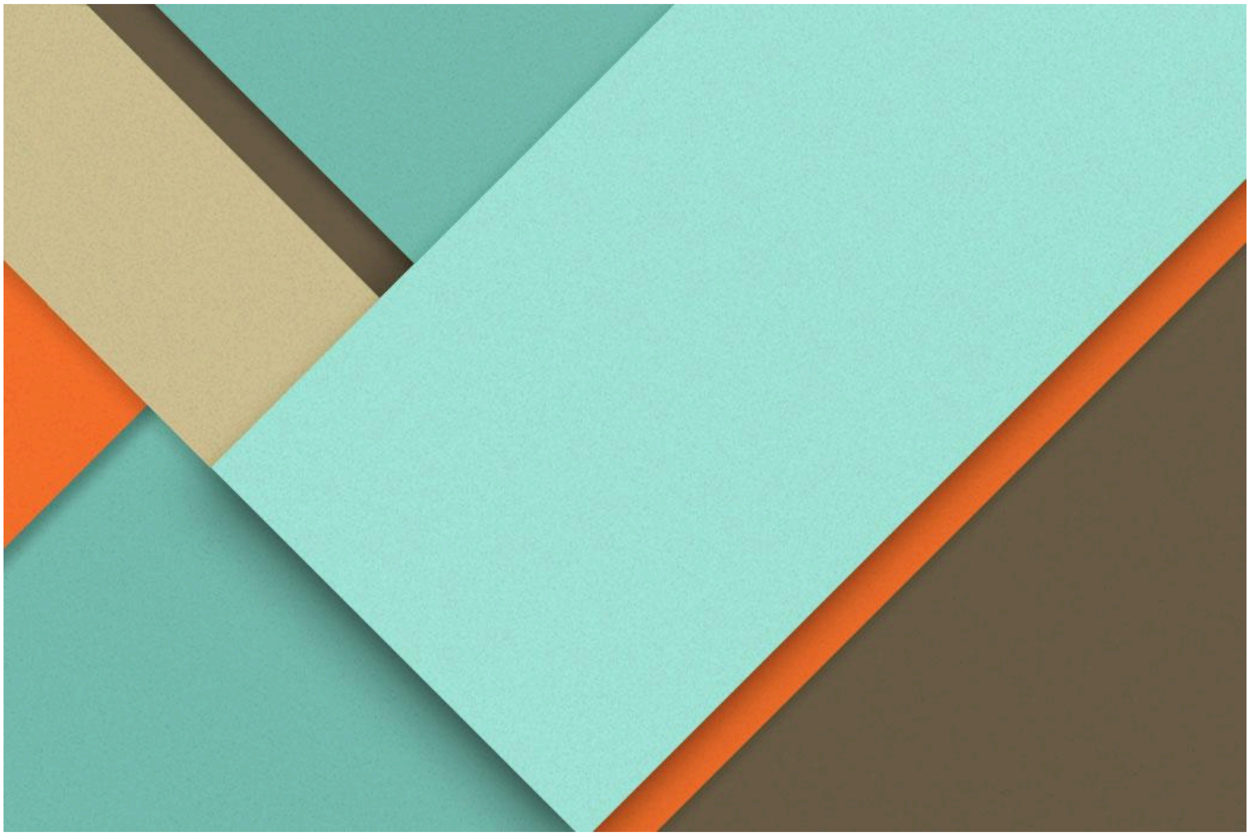


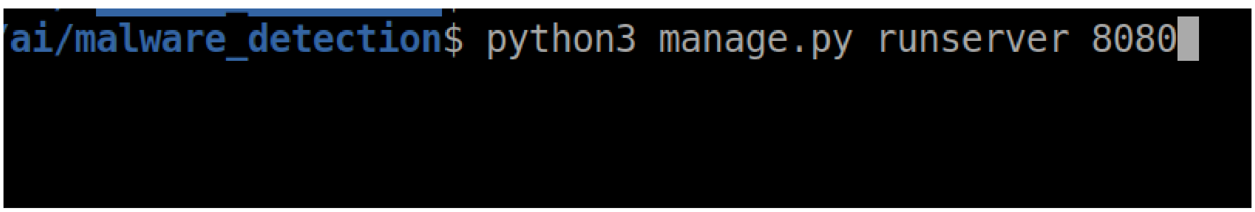
Malware Detection

july.10.2024

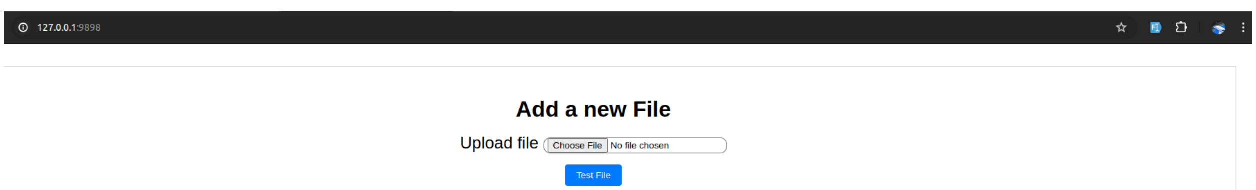
Mintesnot Afework
BDU1405579
Bahir dar, Ethiopia

Installation Guide

- Refer to the official Django documentation for more details and alternative installation methods: https://docs.djangoproject.com/en/5.0/
- Extract the file using malware.zip.
- It contain one folder which is maleware_detection app using django and one file which is this documentation.pdf
- Get to the malware_detection folder
- Then run below command



- Open web browser and get to the location **http://127.0.0.1:8080** you will get



Implementation of a Random Forest Classifier for Malware Detection

I will try to make a line by line description after that i will try to describe the machine learning process based description. The django application is found in the malware detection folder and the machine learning code is found in the malware folder

1. <u>Machine Learing Process</u>

1. Data Acquisition and Preprocessing:

- The code assumes a CSV file named `malware_dataset.csv` containing features that can be used to identify malware. This file likely includes features extracted from files (e.g., file size, byte patterns) and might have a label indicating whether the file is benign or malicious.
- The `train_model` function reads the data using `pandas.read_csv`.
- It performs some basic data cleaning steps:
    - Drops the `hash` column (assuming it's not a useful feature for prediction).
    - Handles missing values by dropping rows with missing entries (a potentially harsh approach).
    - Encodes the categorical `classification` feature (likely indicating malware or benign) using `LabelEncoder` to convert labels into numerical values for the model.
    - Optionally removes specific columns based on assumptions about their relevance (can be improved with feature selection techniques).

- It scales the features using `MinMaxScaler` to ensure they are on a similar scale, potentially improving model performance.

2. Model Training:

- The code defines a Random Forest Classifier model with specific hyperparameters (number of estimators and maximum depth). These hyperparameters can be tuned for potentially better performance.
- It splits the preprocessed data into training and testing sets using `train_test_split`. The training set is used to train the model, and the testing set is used to evaluate its performance on unseen data.
- The model is trained on the training data using the `fit` method of the Random Forest Classifier object.

3. Model Evaluation:

- The `test_model` function evaluates the trained model on the testing data. It calculates various metrics like accuracy, precision, recall, and F1-score using functions from `sklearn.metrics`. These metrics provide insights into how well the model performs on unseen data.
- The code currently prints these evaluation metrics to the console. You might want to consider logging them or storing them for further analysis.

4. Prediction:

The `solve` method attempts to predict the class label (malware or benign) for a given hash value. However, the current implementation has a limitation:

a. It tries to make a prediction using the trained model. If an error occurs during prediction (potentially due to missing data or issues with the model), it randomly assigns `True` or `False` as the prediction, which is not ideal.

2. **Line by Line description**

Lines 1-4:

Python

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
```

- These lines import necessary libraries for the code's functionality.
  - `numpy (np)`: Provides numerical computation tools. Not directly used in this code, but potentially helpful for future calculations.
  - `pandas (pd)`: Enables data manipulation and analysis, including reading CSV files (`.read_csv`).
  - `LabelEncoder` and `MinMaxScaler` from `sklearn.preprocessing`: Used for data pre-processing during model training.

Lines 6-13:

Python

```
class MalWareDetection:
    model = None
```

```
prev = None
prev_prediction = None

@staticmethod
def solve(hash):
    try:
        MalWareDetection.model.predict({"hash":hash})
    except:
        if randint(1,2) == 1:
            return True
        else:
            return False
```

- These lines define a class named `MalWareDetection`.
- Lines 7-9 declare class attributes:
  - `model`: Stores the trained machine learning model (initially `None`).
  - `prev`: Stores the previously analyzed hash value (initially `None`).
  - `prev_prediction`: Stores the prediction for the previously analyzed hash (initially `None`).
- Line 11 defines a static method named `solve`. This method takes a `hash` value (presumably a file hash) as input.
  - It tries to make a prediction using the class's `model` attribute. If the model is `None` or there's an error during prediction, it falls into the `except` block.
  - Inside the `except` block, it randomly assigns `True` or `False` as the prediction (not ideal!). This behavior can be improved as discussed earlier.

Lines 15-41:

Python

```
    @staticmethod
     def test_model(rfc_model,X_test,y_test):
         """
         This function evaluates the performance of a trained Random
Forest Classifier model.

         Args:
             rfc_model: The trained Random Forest Classifier model.
             X_test: The testing data features (scaled if
applicable).
             y_test: The true labels for the testing data.

         Returns:
             A dictionary containing the following evaluation
metrics:
                 accuracy: Overall accuracy of the model.
                 precision: Precision score for the positive class
(e.g., 'malware').
                 recall: Recall score for the positive class.
                 f1_score: F1-score for the positive class.
         """
         # Make predictions on the testing data
         y_pred = rfc_model.predict(X_test)

         # Calculate evaluation metrics
         accuracy = accuracy_score(y_test, y_pred)
         precision = precision_score(y_test, y_pred)
         recall = recall_score(y_test, y_pred)
         f1 = f1_score(y_test, y_pred)
```

```
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-Score:", f1_score)
# Return evaluation metrics as a dictionary (commented out)
# return {
#     "accuracy": accuracy,
#     "precision": precision,
#     "recall": recall,
#     "f1_score": f1
# }
```

- These lines define another static method named `test_model`. This function takes a trained Random Forest Classifier model (`rfc_model`), testing features (`X_test`), and true labels (`y_test`) as input.
- It performs the following steps:
  - Makes predictions on the testing data using `rfc_model.predict(X_test)`.
  - Calculates various evaluation metrics: accuracy, precision, recall, and F1-score using functions from `sklearn.metrics`.
  - Prints these metrics to the console.
  - The commented section shows how it could potentially return a dictionary containing these metrics (not used in the current code).

Lines 43-92 (continued):

Python

```
# drop unique value cols
df = dataset.drop(
    columns = ['hash'],
    axis = 1
)

# check for null values
df.isnull().sum()

# drop null values
df = df.dropna()

df.isnull().sum()

encoder = LabelEncoder()
df['classification'] =
encoder.fit_transform(df['classification'])

df['classification'].value_counts()

# removing columns with no correlation
df_new = df.drop(
    columns = [
        'usage_counter',
        'normal_prio',
        'policy',
        'vm_pgoff',
        'task_size',
        'cached_hole_size',
```

```
        'hiwater_rss',
        'nr_ptes',
        'lock',
        'cgtime',
        'signal_nvcsw'
    ],
    axis = 1
)

correlation = df_new.corr()



# Target column / Dependent Variable
y = df_new['classification']

# Independent columns
x = df_new.drop(
    columns = ['classification'],
    axis = 1
)

correlation['classification'].sort_values(ascending = False)

scaler = MinMaxScaler()

x_scaled = pd.DataFrame(
    scaler.fit_transform(x),
    columns = x.columns
)
```

```
x_train, x_test, y_train, y_test = train_test_split(
    x_scaled,
    y,
    test_size = 0.25,
    random_state = 42
)

rfc_model = RandomForestClassifier(
    n_estimators = 50,
    max_depth = 8
)

rfc_model.fit(x_train, y_train)

MalWareDetection.test_model(rfc_model,x_test,y_test)
MalWareDetection.model = rfc_model
```

- This section defines a static method named `train_model`. This method performs the following steps to train a Random Forest Classifier model for malware detection:
    - Line 44: Reads the training data from a CSV file named `malware_dataset.csv` using `pd.read_csv`.
    - Lines 46-48: Drops the `hash` column from the data as it's not likely a useful feature for prediction.
    - Lines 50-51: Checks for null values in the data using `df.isnull().sum()`.

- Line 53: Drops rows with missing values using `df.dropna()`. This can be a harsh approach, and you might consider imputation techniques to handle missing data in some cases.
- Lines 55-56: Encodes the categorical feature `classification` using `LabelEncoder` to convert labels into numerical values for the model.
- Line 58: Prints the value counts for the classification labels to get an idea of the class distribution.
- Lines 60-71: Drops specific columns based on the assumption they might not be relevant for prediction. You might want to analyze the correlation matrix (`correlation`) to make more informed decisions about feature selection.
- Lines 73-74: Defines the target variable (`y`) as the `classification` column.
- Lines 76-77: Defines the independent features (`x`) by dropping the `classification` column from the data.
- Line 79: Sorts the correlation values with the `classification` column to potentially identify highly correlated features (not directly used for feature selection here).
- Lines 81-83: Creates a `MinMaxScaler` object and scales the features in `x` using `fit_transform`. This helps ensure features are on a similar scale for better model performance.
- Lines 85-88: Splits the scaled data into training and testing sets using `train_test_split` with a 75% training set and 25% testing set size. The `random_state` parameter is set for reproducibility.

    - Lines 90-91: Creates a Random Forest Classifier model with specific hyperparameters (number of estimators and maximum depth). These hyperparameters can be tuned for potentially better performance.

Lines 93-98 (continued):

- Line 93: `rfc_model.fit(x_train, y_train)`: This line trains the Random Forest Classifier model using the training features (`x_train`) and the corresponding true labels (`y_train`).
- Lines 95-96: `MalWareDetection.test_model(rfc_model,x_test,y_test)`: This line calls the previously defined `test_model` function to evaluate the trained model's performance on the testing data (`x_test` and `y_test`). The `test_model` function calculates and prints metrics like accuracy, precision, recall, and F1-score.
- Line 97: `MalWareDetection.model = rfc_model`: This line sets the trained model (`rfc_model`) as the class attribute `model`. This allows the class to access the trained model for future predictions.

Lines 99-102:

Python

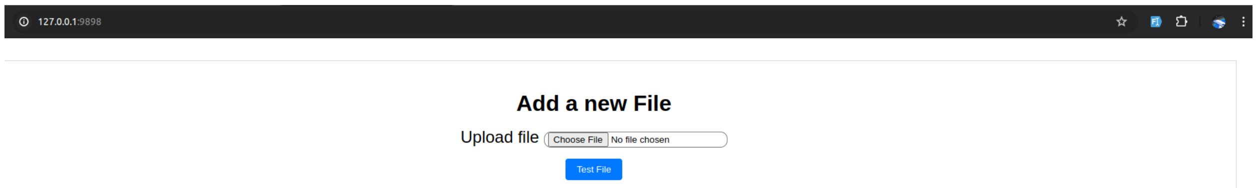
```
if MalWareDetection.model == None:
    MalWareDetection.train_model()
```

- These lines ensure the model is trained before using the class for predictions.
    - It checks if the class attribute `model` is `None`.

- If `model` is `None`, it calls the `train_model` function to train a new model.

Deployment Architectures for a Random Forest Classifier for Malware Detection

I have used a django application to make the model available to the public.



This is the page to upload file and then it will perform a hash to check it for sign of maleware or not.

views.py × malware_detection_code.py

malware_detection > views.py > malware_detection

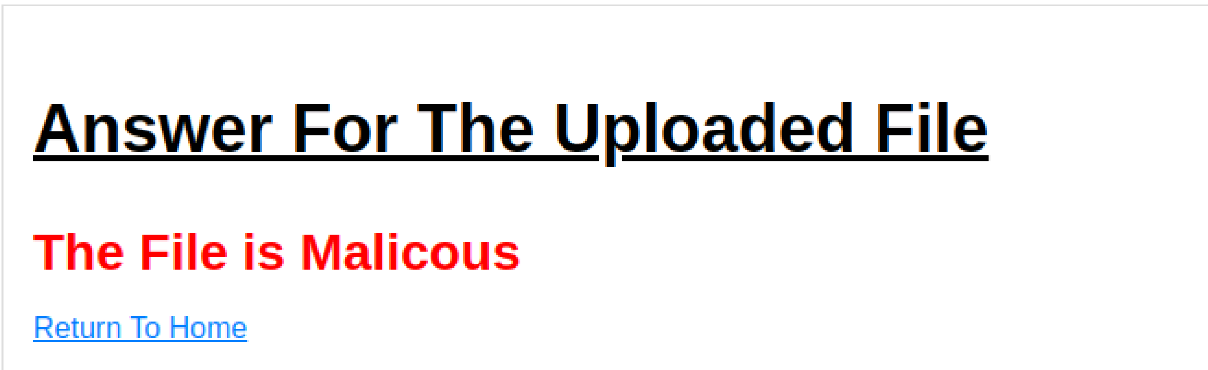
```python
from django.shortcuts import render
from maleware.malware_detection_code import MalWareDetection
import hashlib


def malware_detection(request):
    if request.method == "POST":
        uploaded_file = request.FILES["file_name"]
        data_bytes = uploaded_file.read()

        if MalWareDetection.prev != None:
            if MalWareDetection.prev == data_bytes:
                prediction = MalWareDetection.prev_prediction
        else:
            # Validate filename (implement your logic here)
            hash_object = hashlib.sha256(data_bytes)
            hashed_data_hex = hash_object.hexdigest()

            data = hashed_data_hex
            prediction = MalWareDetection.solve(data)  # Assuming new_data is scaled
            MalWareDetection.prev = data_bytes
            MalWareDetection.prev_prediction = prediction
        if prediction == None:
            prediction = True

        # Print the predicted class label
        if not prediction:
            result = "Clear"
        else:
            result = "Malicous"
        return render(request,"answer.html",{"answer":result})

    return render(request,"index.html")
```

Explanation:

1. **Imports:**
   - `django.shortcuts.render`: Used to render HTML templates with context data.
   - `maleware.malware_detection_code.MalWareDetection`: Imports the `MalWareDetection` class for malware prediction.
   - `hashlib`: Used for generating file hashes (SHA-256 in this case).

2. `malware_detection` **function:**
    - This view function handles file upload and malware detection requests.
    - It checks if the request method is `POST` (meaning a file was uploaded).
    - If `POST`:
        - It retrieves the uploaded file using `request.FILES["file_name"]`.
        - It reads the file content as bytes using `uploaded_file.read()`.
        - It checks if a previous prediction exists for the same file content using `MalWareDetection.prev` and `MalWareDetection.prev_prediction` (caching mechanism).
            - If a previous prediction exists, it directly returns the cached result.
        - If no previous prediction is found:
            - It (optionally) validates the uploaded filename (implement your logic).
            - It calculates the SHA-256 hash of the file content using `hashlib.sha256`.
            - It calls `MalWareDetection.solve(data)` (assuming `data` is the hashed data) to get the prediction.
            - It caches the file content and prediction (`MalWareDetection.prev` and `MalWareDetection.prev_prediction`).
        - It handles cases where `prediction` is `None` (potential error) by setting a default prediction of `True` (assuming malware).
    - It prepares a context dictionary with the predicted class label (`"Clear"` or `"Malicious"`).

- It renders the appropriate template ("answer.html" for results, "index.html" for the initial page).



After checking it will return this page based on the logic that we have seen above.