Eötvös Loránd University

Faculty of Informatics

Department of Software Technology

Mastodon social media threat alert extension

Supervisor:
Zoltán Gera
tanársegéd, programozó matematikus MSc

Author:
Kajdomcaj Dion
Computer Science BSc

Budapest, 2022

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:

Student's Name: Kajdomcaj Dion

Student's Neptun code: E9TZ67

Course Data:

Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Gera Zoltán*

Supervisor's Home Institution: *ELTE Faculty of Informatics*

Address of Supervisor's Home Institution: *1117 Pázmány Péter stny 1/C*

Supervisor's Position and Degree: *egyetemi tanársegéd, programozó matematikus MSc*

Thesis Title: Mastodon Social Media Threat Alert Extension

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Social media thread alert extension is an extension which is planned to improve the security of Mastodon social media platform by letting the users know when there is a possibility for a threat towards them in this social media. This extension will provide social media threat alert for the users, for example when an account texts you it will detect and let you know whether that account has a possibility of being a fake account and afterwards will allow you to decide whether you want to be still connected with that account or not. Besides this it will prevent these kind of users getting the attention they seek in this social media and by that it will start to get them off the network since they will not get the attention they seek for in it, other than this it will encourage students and professors to use their university social media even more than they actually do.

Budapest, 2021. 11. 30.

0.1 Acknowledgment

First of all I want to take the opportunity to thank my family and friends, who supported me continuously during this difficult and stressful journey.

I also want to thank my supervisor Zoltán Gera for motivating me, answering fast to my questions and valuing my work.

For the knowledge gained throughout this journey, the credit goes to my professors, who even in the most difficult pandemic time tried their best to always deliver the material properly and very clearly to us, students.

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays we all know someone who has been affected negatively by the bullying happening on social media, which is becoming a very common problem in our society. Sometimes those cases escalated to the point where it turned into a serious mental health problem for the ones who have been affected. I was always wondering for a way to help people prevent similar cases and feel safe while surfing the social media feed. Everyone has it's rights to feel safe while doing a certain activity, but bringing safety to the social media in these kind of cases has been neglected throughout these last years.

The main reasons mentioned above have pushed me to brainstorm an idea to help people feel safe on Mastodon, which is an open source social media, and enjoy the beauty of connecting with real profiles and use the social media to the point where it changes their life for good.

1.2 Thesis structure

This thesis consists of 4 chapters, which help users understand the installation steps of the software, the correct way of using the software and the possible behavior that can lead to errors, as well as help developers understand the functionality behind the software, the software architecture and the testing made on the software.

Besides those main chapters it also contains a bibliography, a list of figures, a list of codes and a list of tables.

Chapter 2

User documentation

In this chapter we will discuss the installation steps, the correct way of using the software and a brief description about the software.

2.1 Project Description

My project is a desktop application that is meant to run on the background while using Mastodon social media. The whole project was built using the latest *Python* 3 version. The main goal of the project is to detect possible threats coming from other accounts in form of direct messages and tags, after warning the user about possible threats it let's the user decide the kind of action he wants to take against the account that may be a threat and the domain where the account came from.

This project is targeting all the ELTE server accounts, but with few modification it will work for every server.

As a prerequisite for using this desktop application is a stable internet connection and a Mastodon account.

2.2 Installation guide

As earlier mentioned, in order to use Mastodon social media threat alert application we need to have a stable internet connection and a Mastodon account in any server and *Python* 3.10.2.

The application is currently supporting Windows and Linux but the goal is to extend it as a mobile application which supports IOS and Android. Hence, the installation steps are the same for both, Windows and Linux, but we will go through the steps in Windows specifically.

To download the application we need to clone the following repository: `https://github.com/DionKajdomcaj/Mastodon-Social-Threat-Alert.git`.

Prior to cloning the repository we need to make sure that we have git. If git is missing, the user can download it at the following url: `https://git-scm.com/download/win`. If the git prerequisite is met, we can clone the repository by running the following command in the command prompt:

```
C:\Users\dionk>git clone https://github.com/DionKajdomcaj/
    Mastodon-Social-Threat-Alert.git
```

Code 2.1: Cloning Repository

After succeeding to clone the repository we need to install the requirements for our environment. In order to install the requirements we need to make sure we have pip command. If the pip command is not installed, the user can install it by clicking on the following url: `https://phoenixnap.com/kb/install-pip-windows`.

In the application folder there is a text file called requirements, installing it will download all the libraries needed to run the application.

We can install them by using the following command in command prompt:

```
C:\Users\dionk\Mastodon-Social-Threat-Alert>pip install -r
    requirements.txt
```

Code 2.2: Installing requirements

Now we are ready to run the application.

2.2.1 Running the application

In order to run the application we need to make sure that we are in the correct directory and then run the following command:

```
C:\Users\dionk\Mastodon-Social-Threat-Alert>python ThreatAlert.py
```

Code 2.3: Running the application

After running the command, if all the prerequisites are met, we can see our application log in page, but the user must not close the command line while using the application since it will terminate the application.

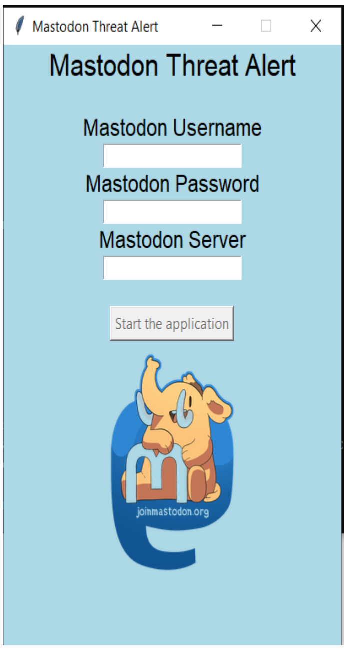


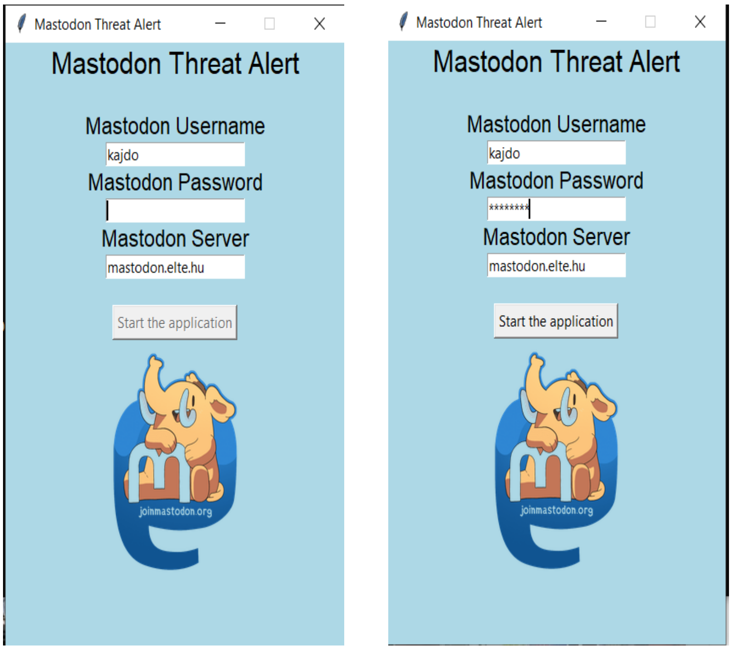
Figure 2.1: Application main page

2.3 Logging in

As we saw in figure 2.1 the button to actually start the application is disabled. In order to enable it we must fill all the necessary text fields which are:

- **Mastodon username**, which is your original Mastodon account username.
- **Mastodon password**, which is your original Mastodon account password, that you use to log in to Mastodon.
- **Mastodon server**, which is the server your account is currently registered in.

If only one of the text fields is missing then the user will not be able to start the application nor use it.



Mastodon Threat Alert

Mastodon Threat Alert

Mastodon Username: kajdo

Mastodon Password:

Mastodon Server: mastodon.elte.hu

Start the application

Mastodon Threat Alert

Mastodon Threat Alert

Mastodon Username: kajdo

Mastodon Password: ********

Mastodon Server: mastodon.elte.hu

Start the application

(a) Missing password field - Button disabled

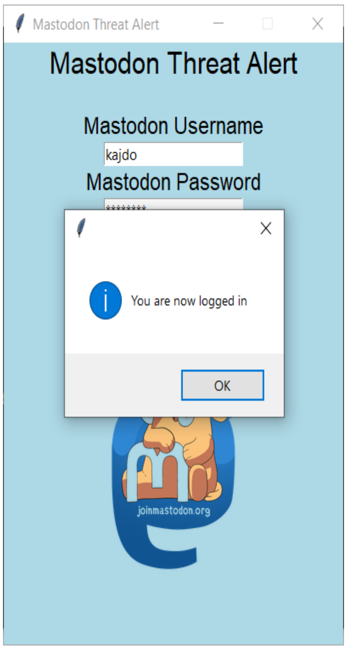
(b) No missing field - Button enabled

Figure 2.2: Button disabled and enabled

The text fields must be filled with the Mastodon account information. In the username field you must enter your Mastodon username, and the same goes up to password. About the server you need to know which server are you in and only type the domain, for example in case there is an account like: `example@mastodon.elte.hu` then the username is **example** and the server is **mastodon.elte.hu**. All of the data are case sensitive, so you must give them exactly as they are originally.

2.3.1 Valid log in data

If every data is correct then the application will be connected to the Mastodon API after we click on the button. But, prior to switching the screen it will show us a confirmation message after API initialization. The API initialization takes a few seconds to be done so the user must wait during those seconds.



Mastodon Threat Alert

Mastodon Username

kajdo

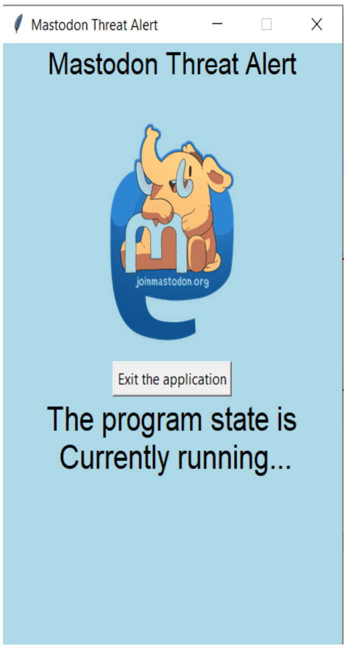
Mastodon Password

****

You are now logged in

Figure 2.3: Confirmation window

After closing the confirmation window, the program will let us know that it is running, and it will not change it's state until it recognizes a possible threat for the logged in user.



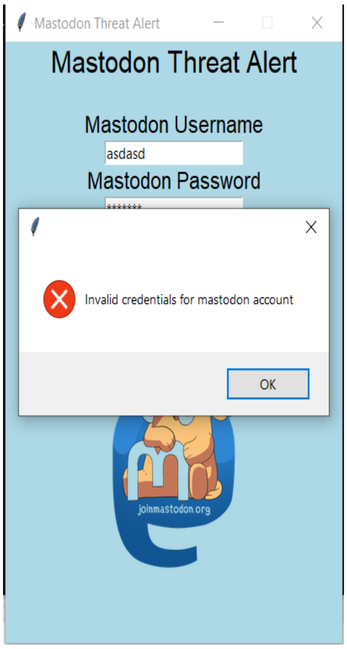
Mastodon Threat Alert

The program state is Currently running...

Figure 2.4: Application running

2.3.2 Invalid log in data

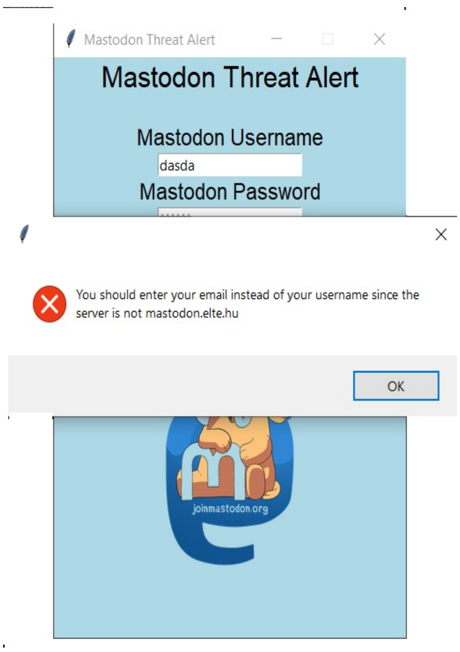
Even if the button is enabled it does not mean that the log in data entered by the user are correct. So, if the user does not give the valid log in data then the application will not be connected to Mastodon API. Hence, it will give the user an error message. The following figure is going to show you the message you will receive for giving invalid log in data. After receiving the message you can just close the message window and try again as many times as you need.



Invalid credentials for mastodon account

Figure 2.5: Incorrect log in data

But if your account is not registered in "mastodon.elte.hu" domain then even if your data is correct you will receive another error message.



You should enter your email instead of your username since the server is not mastodon.elte.hu

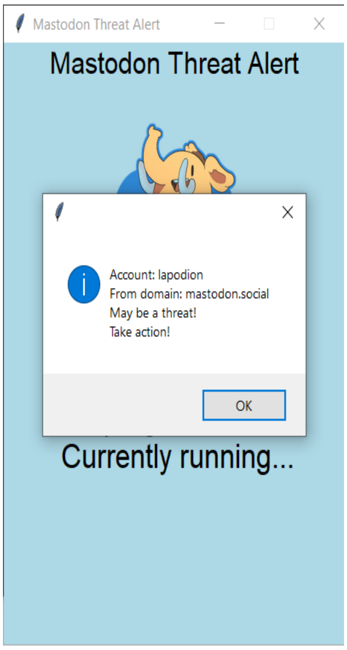
Figure 2.6: User is not in ELTE server

As we can see in 2.6 if the server is not "mastodon.elte.hu" then the email that your mastodon account is registered to should be given instead of your username.

2.4 Actions for the possible threat account

Now that we logged in successfully, we can start using Mastodon as usual, but this time we have the Mastodon threat alert application running on the background and looking for possible threats.

Every time that we are going to receive a direct message or a tag notification, that account's data is going to be checked whether it has a possibility to be a threat or not. After checking if the account, that was trying to reach you, is considered to be a possible threat, the application will show you a warning message containing the possible threat account's username and domain, and will ask you to take a certain action against the account and the account's domain.



Account: lapodion
From domain: mastodon.social
May be a threat!
Take action!

Figure 2.7: Possible threat notification

We have three kind of actions supported in our application depending whether we want to take them against an account or a domain. These actions are:

- **Trust**
- **Block**
- **Mute**

The default value for both of them is Trust, which can be changed.

We have to simply choose the action from a combo box for both, possible threat account and it's domain, and click the button in order to perform the actions.

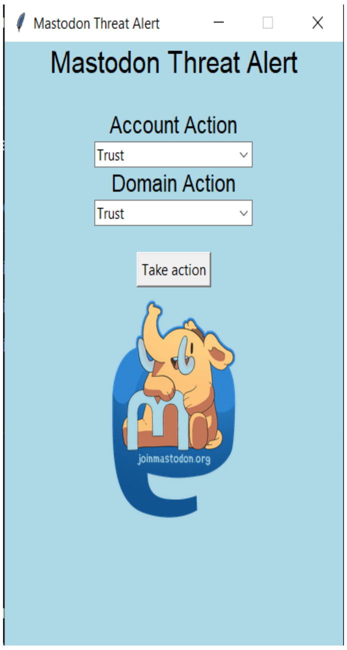
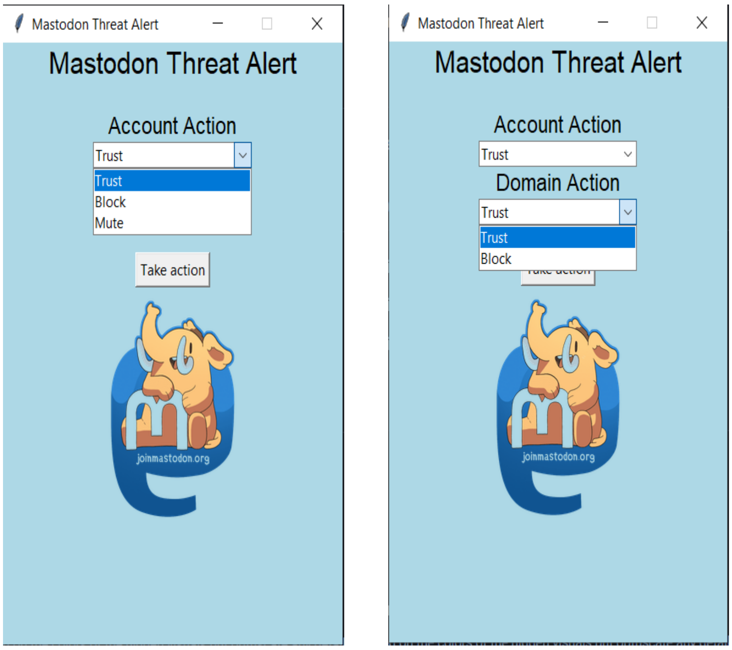


Figure 2.8: Action window

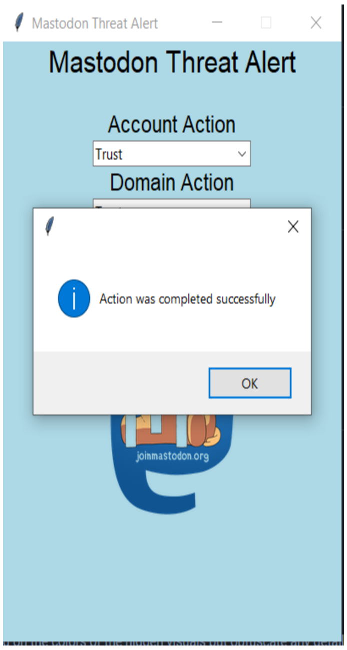
Now we can choose independently the type of action for both, the account and it's domain.



(a) Account Actions (b) Domain Actions

Figure 2.9: Actions available for account and domain

When we click the button, a pop up window will show up, letting you know that the actions were successful.



Mastodon Threat Alert

Account Action

Trust

Domain Action

Action was completed successfully

Figure 2.10: Pop up window to confirm the actions

After closing the pop up window, the application will go back to it's running state as in figure 2.4.

2.4.1 Account action

As it was mentioned before, we can choose the actions separately for the possible threat account and it's domain. In this section we are going to talk about the possible threat account's action.

We have 3 types of actions for the account, which are: Trust, Block, Mute. The following table will describe each action.

Account Actions	
Trust	This action is very simple in context. When the user takes this action it means that he is trusting the possible threat account, and the application will not check the same account if it is trying to reach the user again.

Block	This action blocks the possible threat account, meaning it does not let the account reach the user anymore, not only by direct messaging or tagging, but even following, liking or any other social media activity. Besides those things the user will not see any activity from the possible threat account. To sum it up, the possible threat account will be non existent for the user.
Mute	Muting an account is same as ignoring the account, since the user will not receive notifications from the possible threat account anytime it tries to reach the user, but the user will still be able to check it's activity and the possible threat account will be able to try and reach the user, but simply the user will not be notified.

Table 2.1: Description of every action that can be taken against an account

However, every action that has been taken against the possible threat account can be easily reverted in Mastodon.

2.4.2 Domain action

In case of the possible threat account's domain we have less actions that can be taken against it.

The actions are Trust and Block. They might seem the same as the account actions described in table 2.1, but blocking a domain is completely different, since here we are working with the whole domain. However, in case of Trust, the functionality is the same as it was in the possible threat account's action.

In the following table you can understand their functionality.

Domain Actions	
Trust	This action works the same way as it works to trust an account. However, trusting the domain will not take any actual action against the domain. So, simply the accounts that are from that domain can freely reach the user in any form and without any restrictions.
Block	In case of Block the whole activity of the domain accounts is blocked, meaning that the user won't see any of the accounts, that belong in that domain, activity. But, the user should always be very careful and is recommended not to block domains unless he/she was disturbed many times by the accounts coming from the same domain.

Table 2.2: Description of every action that can be taken against a domain

Same as in the account actions, the domain actions can be reverted in Mastodon.

2.5 Exiting the application

As we can see in figure 2.4, we can exit the application in two ways, by clicking the button named *Exit the application* or by simply closing the window. It is recommended to exit the application by clicking the button because it is terminated safely, but sometimes when we want to exit it while we are not on the running page, we can exit the application by closing the window as well. However, it is not recommended to close the application while taking actions, like in figure 2.8, because the possible threat account's will not be saved in the database and no action will be taken against the possible threat account and it's domain.

Chapter 3

Developer documentation

In this chapter, we are going to get into details about the software architecture, the solutions to certain problems encountered during implementation and how everything came up together in the end.

In order to build and run the application as a developer, you must follow the same steps as in 2.2.

As it was mentioned before the application is completely connected to **Mastodon** [1]. So, it is useless and not the correct way to use it without using Mastodon.

3.1 Mastodon API

As it was mentioned in 2.1, this application checks whether an account that is trying to reach you is a possible threat or not. To do this check we need the data of the account. So, we have to connect to an API that gets the data from the live Mastodon server in order to check the possibility of a threat account. In addition, the API is used to let us take the actions mentioned in 2.4 directly from our application, without the need of doing it through Mastodon. For this application we used an already created API for Python called **Mastodon API** [2].

Below we can see a scheme of how this API operates.

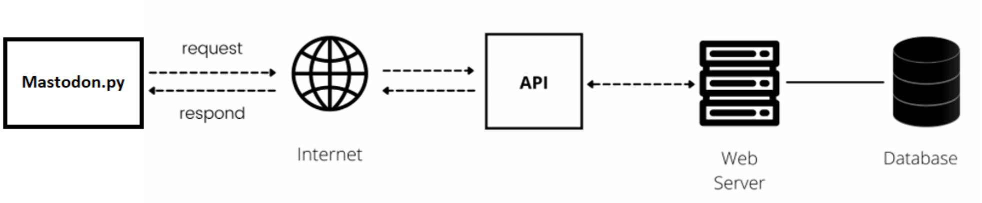


Figure 3.1: Mastodon API scheme

3.1.1 API rate limits

This API has a fixed rate-limit that is 300 requests per 5 minutes, in order not to violate the rate-limit and not make the user wait for 5 minutes until the rate-limit resets, we needed to get data from the live server every 2.6 second so the user can get the full experience of Mastodon. This being said, the user has enough time to surf the feed without having to wait for 5 minutes.

3.1.2 API initialization

To use this API we need to use the user's credentials to log into Mastodon account through the API, and those credentials will be saved in *.secret* files in the directory where we call the API, and that is the reason why the user must give the credentials in the starting window of the application. Below are the methods required to call in order to log into the Mastodon account through the API.

```
def __init__(self):
  self._user = Mastodon()
  self._userApiInstance = Mastodon()
  self._mastodonServer = ""

def createApp(self, mastodon_server):
  if(mastodon_server.split("://")[0] != "https"):
    self._mastodonServer = "https://" + mastodon_server
  else:
    self._mastodonServer = mastodon_server

Mastodon.create_app(
  "mastodonApiAppUser",
  api_base_url=self._mastodonServer,
  to_file='app/secretFolder/mastodonApiAppUser.secret'
)

def setUpAccounts(self):
  self._user = Mastodon(
    client_id='app/secretFolder/mastodonApiAppUser.secret',
    api_base_url=self._mastodonServer
  )

def loginAccount(self, username, password, user=True):
  if user:
    self._user.log_in(
      username,
      password,
      to_file='app/secretFolder/usercredentials.secret'
    )
```

Code 3.1: Logging in to Mastodon account through the API

3.1.3 API instance

Logging in to the account is not enough in order to use this API, but we can say it is a very important step and a must. To actually start using the API we need to create an API instance for that account. Below you can see the method that creates the API instance for both, user account or admin account.

```
def createApiInstance(self):
  self._userApiInstance = Mastodon(
      access_token='app/secretFolder/usercredentials.secret',
      api_base_url=self._mastodonServer
  )
```

Code 3.2: Creating API instance

So, we have to create an access token from the user credentials saved in *.secret* file, and after creating the API instance we can start using the API methods. The main methods that this application uses are:

- **Get the notifications**
- **Clear the notifications**
- **Block an account**
- **Block a domain**
- **Get a certain account's data**
- **Get user's following accounts**
- **Mute an account**

We will deep dive into each of the methods in the following sections.

3.1.4 Notifications

To get the accounts that are trying to reach the user, we first need to get the notifications and filter them to only get the direct messages and tags, which is done by setting the parameter named **mentions_only** to true. To get the notifications through the API we can use the following method:

```
def getNotifications(self):
  return self._userApiInstance.notifications(mentions_only=True)
```

Code 3.3: Getting the notifications by API

And in order to filter them we need to go through each and filter them by their type and existence in the applications database.

```
def startSession(self):
  try:
    notifications = self.api.getNotifications()
    accounts_reaching_user = []
    for notification in notifications:
        account_id = notification['account']['id']

        inDatabase = self.isAccountInDatabase(int(account_id))

        if (account_id not in accounts_reaching_user and not inDatabase):
          accounts_reaching_user.append(account_id)

      return accounts_reaching_user
    except Exception:
      return []
```

Code 3.4: Filtering the notifications

3.1.5 Getting account's data

As it was mentioned in 3.1 we need to get the account's data in order to check for the possibility of the threat. To get the data of a certain account, we need that account's id and we can get the public data for that account such as:

- **Account ID**
- **Username**
- **Domain**
- **Followers Count**
- **Followings Count**
- **Statuses count**
- **Avatar**
- **Header**
- **Date of creation**

And in order to fetch those data we can simply call the API method which is as below:

```
getAccountData(self, account_id, admin=False):
  if not admin:
    return self._userApiInstance.account(account_id)
```

Code 3.5: Fetching certain account's public data

This method will return the data as a dictionary where the keys are the public data names and the values are their values.

3.1.6 Getting user's following accounts

We know that if we follow someone that means we most probably know him/her. Hence, we need to trust the accounts we follow when we start our application. To trust them we need to get the list of the accounts we follow, which is done by the following method:

```
def getFollowingAccounts(self):
  my_id = self._userApiInstance.me()['id']
  return self._userApiInstance.account_following(my_id)
```

Code 3.6: Method to get the list of the accounts we follow

As we can see, we first need to get our account id and then get the list of the accounts we follow. After having the list of the accounts we follow, we need to trust by inserting each in the application's database so we don't have to check them if they try to reach us.

```
def trustFollowings(self):
  following_accounts = self.api.getFollowingAccounts()
  for account_data in following_accounts:
    try:
      account_id = int(account_data['id'])
      username = str(account_data['username'])
      domain = str(account_data['url'].split('/')[2])
      self.__database.insertAccount(account_id, username, domain, False)
      self.__database.insertDomain(domain, False)
    except Exception:
        print("Error")
```

Code 3.7: Inserting the accounts we follow immediately in the application database

3.1.7 Blocking an account

As it was mentioned in 2.4.1 we have the option to block an account and of course the only way to do it outside the application is by using the API. So, in order to block an account we need to call the API method and pass the account's id as a parameter and the account with that id will be blocked.

```
def blockAccount(self, account_id):
  self._userApiInstance.account_block(account_id)
  return True
```

Code 3.8: Blocking an account method

3.1.8 Muting an account

Same as blocking an account, when we want to mute a certain account we need to call the API method and pass the account's id as a parameter.

```
def muteAccount(self, account_id):
  self._userApiInstance.account_mute(account_id)
  return True
```

Code 3.9: Muting an account method

3.1.9 Blocking a domain

When it comes to blocking a domain the parameter changes, since the domains are not classified with ids. If we want to block a domain we need to pass the domain's name as a parameter, but keep in mind that the domain is case-sensitive.

```
def blockDomain(self, domain):
  self._userApiInstance.domain_block(domain)
```

```
    return True
```

Code 3.10: Blockin a domain method

3.2 Predictive Model

Classifying the possibility of a threat is done by a machine learning predictive model. To create our model we used the **Scikit-learn** [3] python library, to preprocess the data we used **Pandas** [4] python library and in order to save the model so it doesn't need to go through the training process again we used **Pickle** [5]

Since our problem was to classify whether an account has a possibility of being a threat, we had to choose between the classification algorithms. Considering multiple factors and trying multiple algorithms we chose to go with Logistic regression for these main reasons:

- **Small dataset**, Logistic regression is known to give the best results in small datasets.
- **Low complexity model**, since we had to classify between two classes and we had 19 features to consider, we needed to use a low complexity model.

3.2.1 Data preprocessing

As it was mentioned above we had to deal with data preprocessing in order to make our dataset suitable to train the model. We needed to read the data from a csv file and create a dataset based on the data that was available on that file. The file contained accounts with account's data and it has already classified all of them as threat and not threat. However, we didn't get all the account's data since some of them were not related to our problem, such as username, account id etc. It is worth mentioning that we considered the following account's data only:

- **Followers count**, the number of the accounts that follow the account.
- **Followings count**, the number of the accounts that the account follows.
- **Statuses count**, the number of the statuses the account has posted.
- **Profile picture**, whether the account has a profile picture.

- **Domain**, account's Mastodon domain.
- **Year of creation**, the year when the account was created.

As we can see we had some of the data in string format, like year of creation and domain, so we had to use the dummy encoder from Pandas and that limited our range of domains and year of creation. So, now the model recognized only the following domains:

- **mastodon.elte.hu**
- **mastodon.social**
- **mastodon.technology**
- **mastodon.xyz**
- **scholar.social**
- **fosstodon.org**
- **hofelho.hu**

About the year range, we thought that it was more than enough to consider accounts created from 2015, since the older the account, the less the possibility of being a threat is. So, after applying the dummy encoder our features increased from 6 to 19.

Since some of our data could get big values, like the followers count and following count, we needed to apply normalization on data, so that we could get more accurate predictions in the end. After normalization we split the dataset into training set and testing set, where 70 percent of the data was on the training set and the other 30 percent of the data was on the testing set.

3.2.2 Exploratory data analysis - EDA

During our EDA we found out that we have some outliars in our dataset, but the number was quite low, somewhere from 4 to 7 in total. Hence, we didn't remove the outliars since the number was very low and it didn't affect the training dataset.

Besides outliars, we didn't have any null value in the fields we used to train, which made our EDA process less complex since we didn't have to deal with null values.

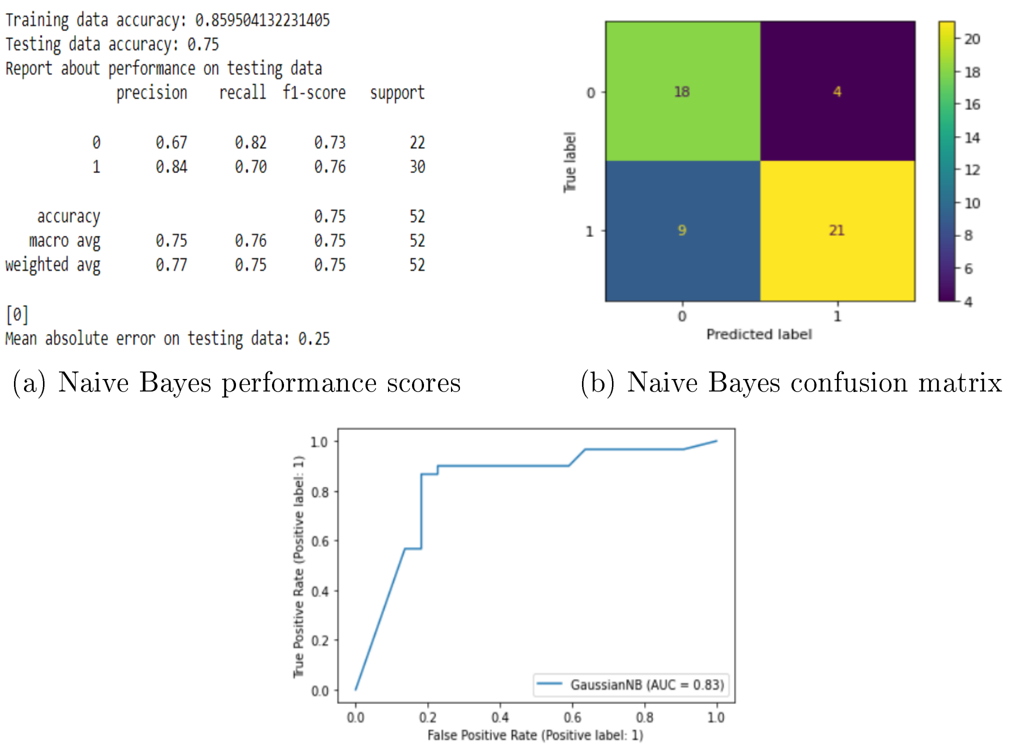
One important thing that is worth mentioning is that we discovered a pattern that showed us that if an account has way more followings than followers, doesn't have a profile picture or the followings count is very low, ranging from 0 to 4, the chances for that account to be a possible threat were higher.

3.2.3 Model performance

We mentioned in 3.2 that we needed a low complexity algorithm, and we know that we have three different kinds provided by Scikit-learn which are:

- **Logistic regression**
- **SVC**
- **Naive Bayes**

After trying each of the algorithms, we wanted to go with Logistic regression since the performance, confusion matrix and the ROC curve was showing us that Logistic regression is performing better in comparison to the other two.



```
Training data accuracy: 0.859504132231405
Testing data accuracy: 0.75
Report about performance on testing data
              precision    recall  f1-score   support

           0       0.67      0.82      0.73        22
           1       0.84      0.70      0.76        30

    accuracy                           0.75        52
   macro avg       0.75      0.76      0.75        52
weighted avg       0.77      0.75      0.75        52

[0]
Mean absolute error on testing data: 0.25
```

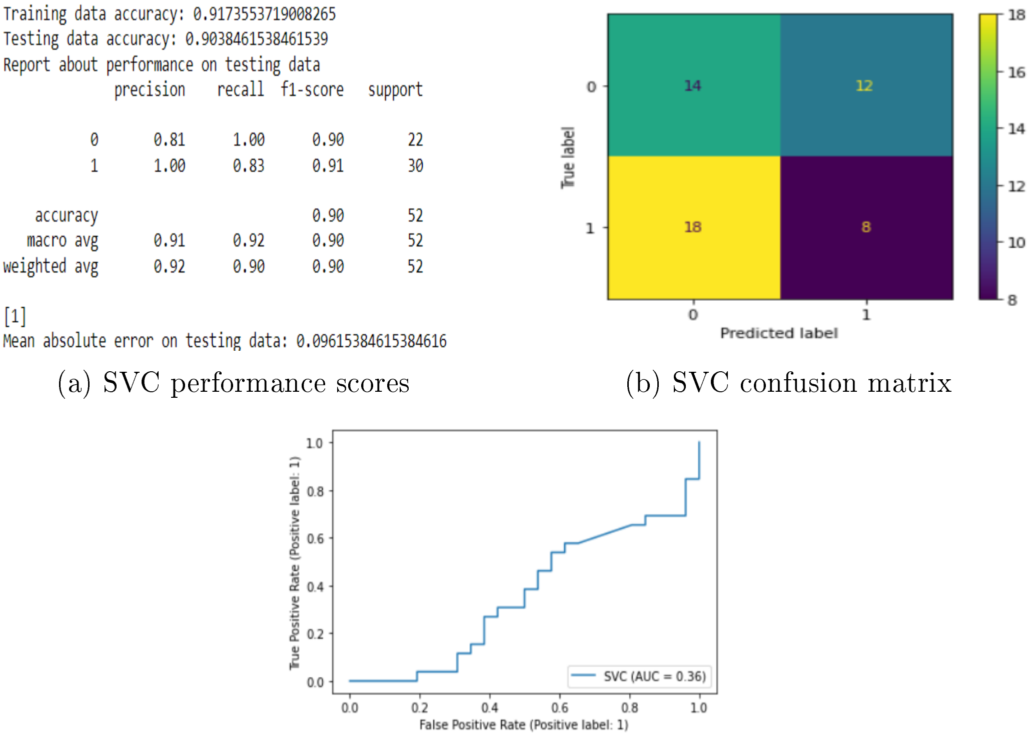
(a) Naive Bayes performance scores

(b) Naive Bayes confusion matrix

(c) Naive Bayes ROC curve

Figure 3.2: Naive Bayes overall performance

For the Naive Bayes algorithm we can see that the mean absolute error is quite high, the difference between the training set accuracy and testing test accuracy is more than 10 percent, the f1-score is quite low and the ROC curve isn't the best.



```
Training data accuracy: 0.9173553719008265
Testing data accuracy: 0.9038461538461539
Report about performance on testing data
              precision    recall  f1-score   support

           0       0.81      1.00      0.90        22
           1       1.00      0.83      0.91        30

    accuracy                           0.90        52
   macro avg       0.91      0.92      0.90        52
weighted avg       0.92      0.90      0.90        52

[1]
Mean absolute error on testing data: 0.09615384615384616
```

(a) SVC performance scores

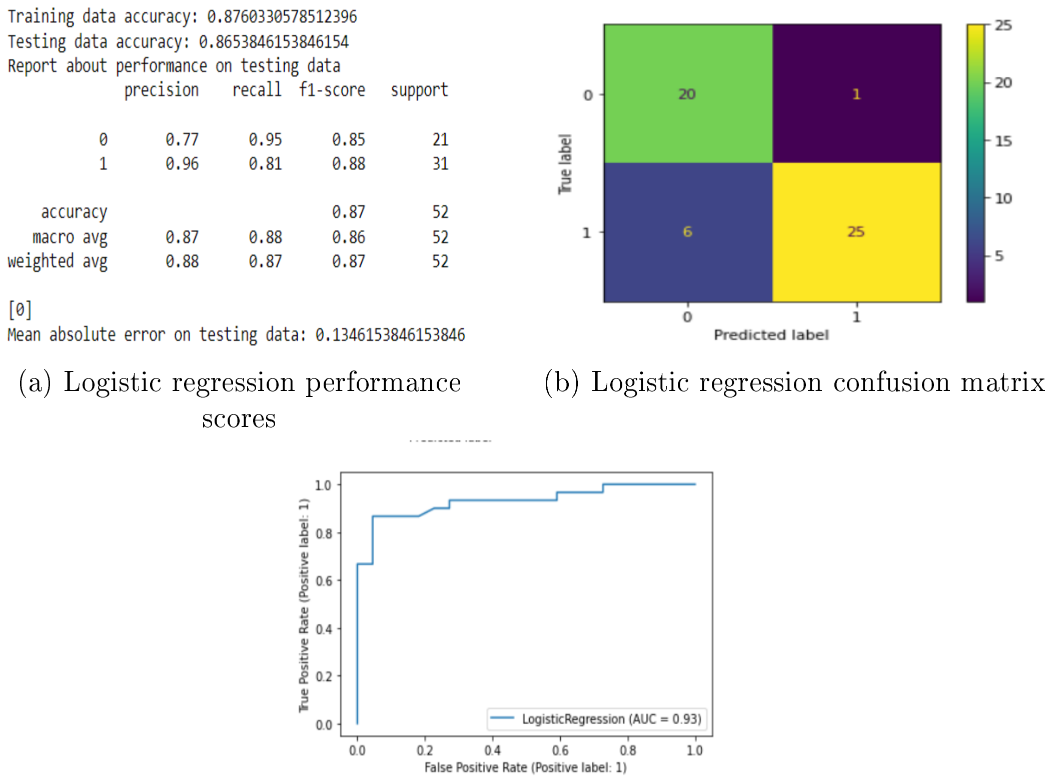
(b) SVC confusion matrix

True Positive Rate (Positive label: 1)
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate (Positive label: 1)
SVC (AUC = 0.36)

(c) SVC ROC curve

Figure 3.3: SVC overall performance

Now for the SVC algorithm we can see perfect scores in case of precision, which is impossible to be 1.0 , meaning that we might have an overfitting problem. But, other than precision the other scores show us a promising model until we watch the confusion matrix and the ROC curve. We can see that the confusion matrix is giving us more false predicted values than it should have, meaning that most of the predictions the model made were false predictions. Not only confusion matrix is giving us a red light for this model, but even the AUC value which is showing us an awful result, where most of the points are below the middle point.



```
Training data accuracy: 0.8760330578512396
Testing data accuracy: 0.8653846153846154
Report about performance on testing data
              precision    recall  f1-score   support

           0       0.77      0.95      0.85        21
           1       0.96      0.81      0.88        31

    accuracy                           0.87        52
   macro avg       0.87      0.88      0.86        52
weighted avg       0.88      0.87      0.87        52

[0]
Mean absolute error on testing data: 0.1346153846153846
```

(a) Logistic regression performance scores

(b) Logistic regression confusion matrix

True Positive Rate (Positive label: 1)
False Positive Rate (Positive label: 1)
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
LogisticRegression (AUC = 0.93)

(c) Logistic regression ROC curve

Figure 3.4: Logistic regression overall performance

However, if we take a look at the Logistic regression algorithm performance then we can see a good accuracy on both, training set and testing set, with a difference of 1.1 percent. Besides the accuracy we see a good recall and precision score, which makes a good f1 score. The confusion matrix tells us that the most of the values predicted are true, and the ROC curve shows us a very good AUC score of 0.93.

3.2.4 Passing account's data to model

As it was mentioned in 3.1.5, the function responsible for fetching data returns a dictionary of multiple data fields for the account. In order to pass the data we first need to filter them and get the fields that are relatable to our model.

```
def modelDecision(self, account_data):
    dataForModel = {}
    dataForModel['followers'] = int(account_data['followers_count'])
    dataForModel['followings'] = int(account_data['following_count'])
    dataForModel['statuses'] = account_data['statuses_count']
    dataForModel['profile'] = 1 if 'missing' not in account_data['
       avatar'].split('/') else 0
    dataForModel['server'] = account_data['url'].split('/')[2]
    dataForModel['dateOfCreation'] = int(str(account_data['
       created_at']).split('-')[0])
    data = []
    for index, feature in enumerate(self.features):
      if index < 4:
        data.append(dataForModel[feature])
      elif index > 3 and index < 11:
        if feature == dataForModel['server']:
          data.append(1)
        else:
          data.append(0)
      else:
        if feature == dataForModel['dateOfCreation']:
          data.append(1)
        else:
          data.append(0)

    model_result = bool(self.model.predict([data])[0])
    print(model_result)
    return model_result
```

Code 3.11: Function that returns the model decision

The function **modelDecision** filters the data to the ones that are relatable to our model and afterwards uses those data to predict the result, which is type converted from integer to boolean, and then returned as a boolean in the end.

3.3 Database

We used database in our application in order to save the already checked users and domains. For the database we chose to use the **SQLite** [6] with it's API python library called **sqlite3** [7].

We created two tables, one for the handled accounts and one for the handled domains. They are both initialized at the same time and they are not connected with each other, meaning that they behave independently.

As a first step of initializing the database is to connect to it, create the tables and after we are done we need to commit the data.

3.3.1 Handled accounts table

The handled accounts table consists of these data:

- **Account id**, as a primary key.
- **Account username**
- **Account domain**
- **Threat boolean**

So, whenever an account is getting checked for the threat possibility, it will immediately go in the database after the threat possibility result is given, and that account doesn't need to be checked twice since it's in the database of the handled accounts already.

3.3.2 Handled domains table

The handled domain table consists of these data:

- **Domain name**, as a primary key.
- **Threat boolean**

Same as in 3.3.1, we will insert the domain into the handled domain table immediately after receiving the threat possibility result.

3.4 Graphical user interface

This application uses a very simple GUI, with the aim that the user cannot do any big mistake during runtime. Since, it is planned to run on the background we valued the program safety during the runtime more than the design itself.

The library we used for the GUI is **Tkinter** [8], which is provided by python for desktop applications containing many useful elements. In order to read the images we used **Pillow** [9].

Our GUI consists of a front-end and a back-end part, where the front-end is responsible for displaying the state of the action executions and the program. Whereas, the back-end is responsible for the whole logic behind predicting whether an account has the possibility of the threat or not.

It is worth mentioning that the whole GUI is not resizable, meaning that there is one permanent size, and this is done for the reason that the user needs the GUI to be as small as possible but well visible, because he will run it on the background.

During the front-end our main problem was updating the window constantly since our application has some loops and the Tkinter built-in function **mainloop** is a loop itself. Even when we wanted to initialize the API and the database, there were a few seconds taken while being executed, meaning that we had to start a separate thread to initialize it while the window is being updated constantly until the initialization is completed.

The thread was created by using the **Threading** [10] python module.

```python
def initAppReq(self):
  try:
    self.app = Application(self.username, self.password, self.server)
    self.app.initApi()
    self.done_init = True
  except Exception:
    self.widgets['main'][7]['state'] = tk.NORMAL
    if('elte.hu' not in self.server and '@' not in self.username):
      showerror(message="You should enter your email "
                + "instead of your username "
                + "since the server is not "
                + "mastodon.elte.hu")
    else:
      showerror(message="Invalid credentials for mastodon account")

def startApp(self):
  self.widgets['main'][7]['state'] = tk.DISABLED
  self.update()
```

```
    init_thread = Thread(target=self.initAppReq, daemon=True)
    init_thread.start()
    while not self.done_init:
      self.update()
      if self.done_init:
        break
      self.update()
    self.app.initDatabase()
    init_thread.join()
    self.update()
    showinfo(message="You are now logged in")
    self.runningScreen()
    self.protocol('WM_DELETE_WINDOW', self.stopApp)
    self.callSession()
```

Code 3.12: Updating window while initializing the API and database

As well as some of our API calls required a waiting time as mentioned in 3.1.1, so we again scheduled them and made an inner loop that updated the window constantly until the API call was executed.

```
  def callSession(self):
    while True:
      self.update()
      print("running")
      try:
        self.accounts_reaching_user = []
        self.after(2400, self.sendRequest)
        self.waitTime(2.6)

        if len(self.accounts_reaching_user) > 0:
          for account in self.accounts_reaching_user:
            self.update()
            threat_checked_account = self.app.isItThreat(account)
            account_data = threat_checked_account[0]
            account_name = account_data['username']
            threat_data = threat_checked_account[1]
            domain = account_data['url'].split('/')[2]
            if threat_data:
              message = ("Account: " + account_name
                  + "\nFrom domain: " + domain
                  + "\nMay be a threat!\nTake action!")
            showinfo(message=message)

          self.done = False

          self.cleanRunning()
          self.createAction(account_data, threat_data)

          while not self.done:
            self.update()
            if self.done:
              break
          self.cleanAction()
      self.app.insertAccountInDatabase(account_data,
                      threat_data)
    except Exception:
      print("Error")
```

Code 3.13: Updating the window while calling API functions

Our application has three screens in total, which are:

- Starting Screen
- Running Screen
- Action Screen

The approach we took to switch between screens is to draw them when it is needed and clear them if it isn't needed.

All the screen elements are saved in a dictionary created at the GUI initialization where the keys describe the screens and the values are list containing the widgets of each screen.

Below we can see the functions that handles this approach.

```
def runningScreen(self):
  for widget in self.widgets['main']:
    widget.pack_forget()
  for widget in self.widgets['running']:
    widget.pack()
```

Code 3.14: Switching from starting screen to running screen

Switching between the starting screen to the running screen is done by removing the widgets of the starting screen and initializing the ones in the running screen.

```
 def cleanRunning(self):
  for widget in self.widgets['running']:
    widget.pack_forget()

def createAction(self, account_data, threat_data):
  self.canvas.pack_forget()
  self.action_label1 = tk.Label(self, text="Account Action",
                font=('Ariel', 14), bg='light blue')
  self.widgets['action'].add(self.action_label1)
  self.action_label1.pack()

  self.string_var = tk.StringVar()
  self.action_combobox = Combobox(self,
  textvariable=self.string_var)

  self.action_combobox['values'] = ("Trust",
                  "Block",
                  "Mute")
  self.action_combobox.current(0)
  self.widgets['action'].add(self.action_combobox)
  self.action_combobox.pack()

  self.action_label2 = tk.Label(self, text="Domain Action",
                font=('Ariel', 14), bg='light blue')

  self.widgets['action'].add(self.action_label2)
  self.action_label2.pack()

  self.string_var2 = tk.StringVar()
  self.domain_combobox = Combobox(self,
                textvariable=self.string_var2)
  self.domain_combobox['values'] = ("Trust",
                  "Block")
  self.domain_combobox.current(0)
```

```
    self.widgets['action'].add(self.domain_combobox)
    self.domain_combobox.pack()

    self.empty_label3 = tk.Label(self, bg='light blue')
    self.widgets['action'].add(self.empty_label3)
    self.empty_label3.pack()

    self.action_button = tk.Button(self,
                text='Take action',
                command=lambda:
                self.takeAction(account_data, threat_data))

    self.widgets['action'].add(self.action_button)
    self.action_button.pack()
    self.canvas.pack()
```

Code 3.15: Switching from running screen to action screen

Switching from running screen to action screen is done by removing the running screen widgets and creating the elements of the action screen and packing them in the frame.

```
def cleanAction(self):
  for widget in self.widgets['action']:
    widget.destroy()
  self.widgets['action'].clear()
  for widget in self.widgets['running']:
    widget.pack()
```

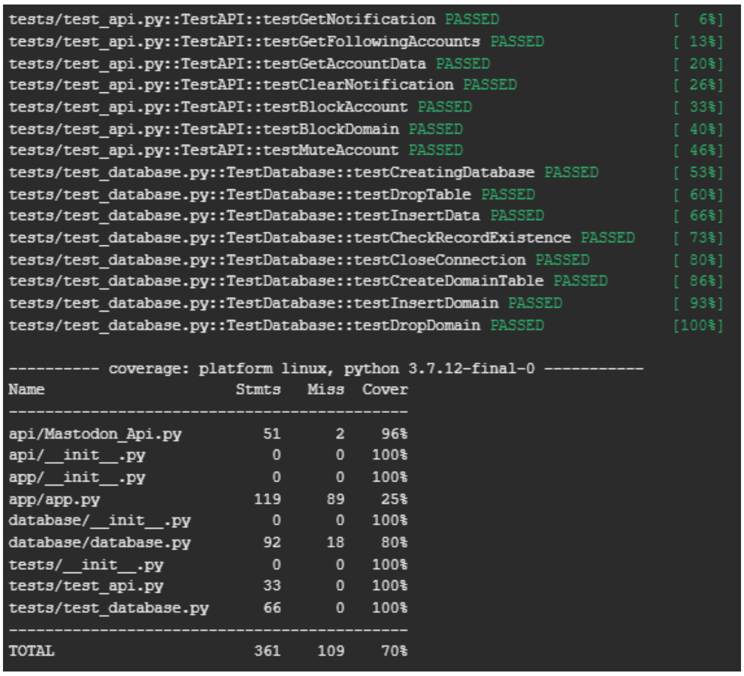
Code 3.16: Switching from action screen to running screen

And the other way around, switching from action screen to running screen is done simply by destroying the widgets as object and clearing the list under which they were saved, and afterwards packing the running widgets to the frame.

3.5 Testing

To test our application we used **unit testing** and **manual testing**. Running the unit tests was done by the **Pytest** [11] framework, but the run was done by the

CI build and not locally, so the test success can be checked in the git repository (see 2.2), under details in CI build.



```
tests/test_api.py::TestAPI::testGetNotification PASSED                  [  6%]
tests/test_api.py::TestAPI::testGetFollowingAccounts PASSED             [ 13%]
tests/test_api.py::TestAPI::testGetAccountData PASSED                   [ 20%]
tests/test_api.py::TestAPI::testClearNotification PASSED                [ 26%]
tests/test_api.py::TestAPI::testBlockAccount PASSED                     [ 33%]
tests/test_api.py::TestAPI::testBlockDomain PASSED                      [ 40%]
tests/test_api.py::TestAPI::testMuteAccount PASSED                      [ 46%]
tests/test_database.py::TestDatabase::testCreatingDatabase PASSED       [ 53%]
tests/test_database.py::TestDatabase::testDropTable PASSED              [ 60%]
tests/test_database.py::TestDatabase::testInsertData PASSED             [ 66%]
tests/test_database.py::TestDatabase::testCheckRecordExistence PASSED   [ 73%]
tests/test_database.py::TestDatabase::testCloseConnection PASSED        [ 80%]
tests/test_database.py::TestDatabase::testCreateDomainTable PASSED      [ 86%]
tests/test_database.py::TestDatabase::testInsertDomain PASSED           [ 93%]
tests/test_database.py::TestDatabase::testDropDomain PASSED             [100%]

---------- coverage: platform linux, python 3.7.12-final-0 -----------
Name                          Stmts   Miss  Cover
-------------------------------------------------
api/Mastodon_Api.py              51      2    96%
api/__init__.py                   0      0   100%
app/__init__.py                   0      0   100%
app/app.py                      119     89    25%
database/__init__.py              0      0   100%
database/database.py             92     18    80%
tests/__init__.py                 0      0   100%
tests/test_api.py                33      0   100%
tests/test_database.py           66      0   100%
-------------------------------------------------
TOTAL                           361    109    70%
```

Figure 3.5: Test coverage and success

Our unit test coverage was 70 percent, and the other 30 are functions that depend on user behavior, so they were tested manually. We applied unit tests on both, API and database, in order to have a quality product.

We had the test stored separately for database and API, but they were both tested outside their own class, because in order to run the tests using Pytest the python file must have the name test in the beginning or in the end, and the same applies to the testing method.

```python
def testCloseConnection(self):
  database = Database()
  database.dropAccountTable()
  database.createTableAccounts()

  id = 1234123
  user = 'Dion'
  domain = 'gibberish'
  threat = False
```

```

    database.insertAccount(id, user, domain, threat)

    assert True == database.closeConnection()

    for _ in range(3):
    assert False == database.closeConnection()
```

Code 3.17: Example method to test a unit

Hence, we did two separate files, one carries the testing of the database and the other one carries the API testing, but both of the files are under the same directory called testing.

We also tested the code style using the **Flake8** [12] library, which checks your code base against the PEP8 coding style.

3.5.1 Manual testing

Some of the units were dependent on user behavior so we manually tested them. In addition, we tested the GUI functionality manually as well, to ensure that we are delivering a fast and a non-crashing GUI. Besides those, the switching between screens was tested, the account information window (see 2.7), action taking fields (see 2.8) in the action screen.

Chapter 4

Conclusion

Knowing that **Mastodon** is evolving day by day, and many officials are starting to use it, I think that a software like **Mastodon Social Media Threat Alert** is needed to ensure safety for the users, and to make the users feel more comfortable while using it, especially when it comes to organization servers, that are a target for scammers, fake profiles etc. It will even increase **Mastodon** users, since there is still no social media guaranteeing you that it filters the fake profiles from the real ones, and in case of organizations, this is very important since they know exactly what they are dealing with. I truly believe that with the right team the project is going to have a very promising future.

The way I think this software can improve is to be integrated in every new server that opens up, so that the users don't need to run it at all, but everything should run internally on the server they are registered in. Another improvement point is the model, where I think it can be even more accurate if we train it with a very large dataset from different servers, which I couldn't do since I had limited access, but the **Mastodon** itself could invest in training the model that way, and making it learn continuously, so that the accuracy could reach a percentage of at least 95 percent. Even the features could be extended, like reporting an account to the server admin so he/she can inspect it manually and even ban the account's ip from the server, if he/she thinks that it is a fake account.

I truly believe that this software is very valuable and needs to be implemented not only in **Mastodon**, but even in other open source social medias, or even other open source software that require accounts in order to use them.

Bibliography

[1] Mastodon org. *Mastodon open source social media.* URL: https://www.joinmastodon.org.

[2] Lorenz Diener. *Mastodon API for Python.* URL: https://github.com/halcy/Mastodon.py.

[3] F. Pedregosa et al. *Scikit-learn: Machine Learning Library in Python.* URL: https://scikit-learn.org/stable/.

[4] Wes McKinney. *Data structures for statistical computing in python.* URL: https://pandas.pydata.org/stable.

[5] Guido van Rossum. *Python object serialization.* URL: https://docs.python.org/3/library/pickle.html.

[6] Dwayne Richard Hipp. *SQL database engine.* URL: https://www.sqlite.org/index.html.

[7] Gerhard Häring. *DB-API 2.0 interface for SQLite databases.* URL: https://docs.python.org/3/library/sqlite3.html.

[8] Steen Lumholt and Guido van Rossum. *Python interface to Tcl/Tk.* URL: https://docs.python.org/3/library/tkinter.html.

[9] Alex Clark. *Friendly Python Imaging Library fork.* URL: https://pillow.readthedocs.io/en/stable/.

[10] Guido van Rossum and other contributors. *Thread-based parallelism.* URL: https://docs.python.org/3/library/threading.html.

[11] Brian Okken. *Python testing tool.* URL: https://docs.pytest.org/en/7.1.x/.

[12] Joe Joyce. *Tool For Style Guide Enforcement.* URL: https://flake8.pycqa.org/en/latest/.

List of Figures

List of Tables

List of Codes