

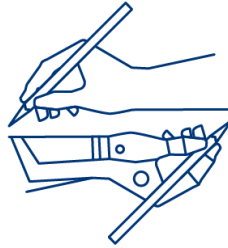


Project: 101094364 — ITHACA — HORIZON-CL2-2022-DEMOCRACY-01

EUROPEAN RESEARCH EXECUTIVE AGENCY (REA)

REA.C – Future Society

C.1 – Inclusive Society



ITHACA

AI To Enhance Civic Participation

ITHACA

artificial Intelligence To enHance Civic pArticipation

D5.2: ITHACA Conformity assessment mechanisms_v1 - prototype

Work Package: WP5 – Conformity assessment tools policy recommendations and guidelines

Authors:	UPAT (Loi, Zachos, Moustakas), CERTH (Spiliotis, Chandrinos, Panou), UniGraz (Zangl, Bedek, Nussbaumer, Weichselgartner, Albert)
Status:	Final
Due Date:	30/6/2024
Version:	1.0
Submission Date:	30/6/2024
Dissemination Level:	PU – Public

Disclaimer:

This document is issued within the frame and for the purpose of the ITHACA project. This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101094364. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

This document and its content are the property of the ITHACA Consortium. All rights relevant to this document are determined by the applicable laws. Access to this document does not grant any right or license on the document or its contents. This document or its contents are not to be used or treated in any manner inconsistent with the rights or interests of the ITHACA Consortium or the Partners detriment and are not to be disclosed externally without prior written consent from the ITHACA Partners. Each ITHACA Partner may use this document in conformity with the ITHACA Consortium Grant Agreement provisions.

(*) Dissemination level. - Public — fully open (automatically posted online)

Sensitive — limited under the conditions of the Grant Agreement

EU classified —RESTREINT-UE/EU-RESTRICTED, CONFIDENTIEL-UE/EU-CONFIDENTIAL, SECRET-UE/EU-SECRET under Decision 2015/444

ITHACA Project Profile

Grant Agreement No.: 101094364

Acronym:	ITHACA
Title:	artificial Intelligence To enHance Civic pArticipation
URL:	https://www.ithaca-project.eu/
Start Date:	01/01/2023
Duration:	36 months

Partners

Short Name	Legal Name	Country
KT	KONNEKT ABLE TECHNOLOGIES LIMITED	IE
CERTH	ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS	EL
UPAT	PANEPISTIMIO PATRON	EL
RtF	RAISING THE FLOOR	BE
SnP	STAMADIANOS KAI SYNETAIROI DIKIGORIKI ETAIREIA	EL
UniGraz	UNIVERSITAET GRAZ	AT
MNLT	MNLT INNOVATIONS IKE	EL
SIMAVI	SOFTWARE IMAGINATION & VISION SRL	RO
PEDAL	PEDAL CONSULTING SRO	SK
BMA	AGENTIA METROPOLITANA PENTRU DEZVOLTARE DURABILA BRAȘOV ASOCIATIA	RO
MARTIN	MESTO MARTIN	SK

Document History

Version	Date	Author (Partner)	Remarks/Changes
0.1	29/03/2024	UPAT	First Structure
0.2	10/05/2024	UniGraz	Executive Summary / Introduction
0.3	14/05/2024	UPAT	Section 4
0.4	17/05/2024	UPAT	Section 5
0.5	20/05/2024	CERTH	Section 6
0.6	30/05/2024	UPAT	Section 7
0.7	10/06/2024	UPAT	Section 2 & Section 3
0.8	11/06/2024	UniGraz	Update Introduction
0.9	15/06/2024	UniGraz & CERTH	Review
1.0	27/06/2024	UPAT	Final Version and Submission to KT

Executive Summary

The ITHACA project develops a platform that aims to enhance civic participation by incorporating Artificial Intelligence (AI) methods and components. This report is the technical complement of the ITHACA Deliverable 5.1 (*'ITHACA Conformity assessment mechanisms_v1 - report'*) that focuses on conformity assessment mechanisms in line with the ethical principles of Fairness, Security, and Privacy. In D5.1 the concepts of four tools have been elaborated that address ethical problems related problematic and toxic content, privacy issues, and cyber-attacks on the models. This deliverable D5.2 outlines the technical development and documentation of these four tools. It describes details of the models' classes for the 4 components and tools dedicated to ensuring the above-mentioned ethical principles:

(i) The *AI fairness tool* for fairness conformity assessment that aims to detect toxic speech related to a vulnerable group: A technical description of the functions sequenced to a pipeline is provided in this report.

(ii) The *Privacy-Preserving Machine Learning (PPML) tool* for privacy conformity in AI-based systems and big data for civic participation applications: This tool consists of a model for Differential Privacy that protects user privacy by adding noise to the privacy model.

(iii) The *AI cybersecurity tool* for security risk and threat detection. This tool employs the open-source tool ModelScan in order to discover security breaches of the AI models used in ITHACA.

(iv) The *Visual component*: This tool enables different stakeholders of the ITHACA platform (in particular administrators, moderators and/or maintainers) to efficiently control the data and by providing visual inspection and conflictive event detection.

Together with its more conceptual 'twin', i.e. D5.1, this report D5.2 represents the first version of the concepts and development of the four tools. The second and updated versions (D5.3 and D5.4) will be available at the End of 2024.

Table of Content

EXECUTIVE SUMMARY	4
TABLE OF CONTENT	5
LIST OF FIGURES.....	6
LIST OF TABLES.....	7
LIST OF ABBREVIATIONS	8
1 INTRODUCTION	9
2 INSTALLATION & USAGE GUIDE.....	10
3 DEMONSTRATION	12
4 AI FAIRNESS TOOL.....	16
5 PRIVACY PRESERVING MACHINE LEARNING TOOL	20
6 AI CYBERSECURITY TOOL.....	23
7 VISUAL COMPONENT	27
8 REFERENCES	30

List of Figures

Figure 1: Common code execution order	10
Figure 2: Loading and browsing platform posts on the Visual Component.	12
Figure 3: Editing selected post from list.	13
Figure 4: Example of potentially toxic words highlighting.	13
Figure 5: Detection of potentially toxic words on custom text written by user.	13
Figure 6: Privacy report based on model resilience against threshold attacks. Resilience is quantified via the metrics Area Under Curve (AUC), Attacker advantage and Positive predictive value, described in Section 2.3.2 of D5.1.	14
Figure 7: Full view of the Visual Component Tool.	15
Figure 8: The models' architecture upon which ModelScan was tested.	23

List of Tables

Table 1 Visual component Features description 14

List of Abbreviations

Abbreviation /acronym	Description
PPML	Privacy Preserving Machine Learning
DP	Differential Privacy
AI	Artificial Intelligence
NN	Neural Network
GUI	Graphical User Interface
AUC	Area Under Curve
PPV	Positive Predictive Value
NLP	Natural Language Processing

1 Introduction

The deliverable D5.2 represents parts of the results and outcomes of the Task 5.1 called ‘Fairness, Security and Privacy conformity assessment mechanisms’. Overall, Task 5.1 will result in 4 Deliverables:

- D5.1 – ITHACA Conformity assessment mechanisms_v1 (M18, Report)
- D5.2 – ITHACA Conformity assessment mechanisms_v1 (M18, Demonstrator)
- D5.3 – ITHACA Conformity assessment mechanisms_v2 (M24, Report)
- D5.4 – ITHACA Conformity assessment mechanisms_v2 (M24, Demonstrator)

Both pairs of documents (i.e. the versions 1 in M18 and version 2 M24) can be considered as complementary ‘twins’. While the reports D5.1 and D5.3 are more conceptual by describing the underlying reasoning based on legal and user requirements, the more technical D5.2 and D5.4 consist of the implementation of the tools and related documentation and the models' classes for the ITHACA components on fairness, security and privacy conformity assessment mechanisms.

We would like to refer to the Introduction Chapter of D5.1 for more details on the purpose and scope of these two complementary deliverables D5.1 and D5.2, as well as their relations to other Deliverables, Work Packages and Tasks.

This deliverable provides technical details of the three tools as well as visual component and is structured as follows:

- In **Chapter 2** a guide with installation and usage instructions of the evaluation tools that were developed throughout the first phase of task T5.1 is provided.
- In **Chapter 3** a demonstration of the functionality of the implemented tools, depicted through Visual Component is provided. This Chapter aims to give the reader an example of the usage of the tools in a civic engagement platform and a level of explainability over the outcomes of these tools.
- The *AI fairness tool* for fairness conformity assessment is presented in **Chapter 4**. It provides documentation of the main functions of the pipeline to evaluate if a text contains toxic speech in short text pieces.
- The *Privacy-Preserving Machine Learning (PPML) tool* for privacy conformity in AI-based systems and big data for civic participation applications is presented in **Chapter 5**. It provides documentation of the functions and scripts that are used to set up a model for the Differential Privacy (DP) method used by this tool.
- The *AI cybersecurity tool* for security risk and threat detection is presented in **Chapter 6**. It describes the tool for analysing and categorising the safety level the AI models in ITHACA.
- The *Visual component* to efficiently control the data and to allow for a visual inspection and conflictive event detection is described in **Chapter 7**. It describes the tool that provides visual oversight functions of the above-mentioned tools and the user interface to their outcomes.

2 Installation & Usage Guide

For the purposes of this deliverable, the source code files are available in this [GitHub Repository](#).

After downloading the files, and installing Python 3.11, the necessary dependencies can be pulled via entering the following on the command prompt:

```
pip install -r requirements.txt
```

After a while the necessary packages will be available, and all scripts will be ready to run. The traditional order by which the scripts are executed are as is shown in Figure 1, however, to avoid unnecessarily long code execution commonly required when training neural networks (i.e. in `train_model.py` and `train_private_model.py`) the necessary output of said code is also included so that the execution of the Visual Component directly is possible via the `visual.py` file.

Figure 1, describes the interconnection between all available source code files. Specifically one-way arrows signify a parent-child relationship, meaning that the output of the parent file (shown by the beginning of the arrow) is necessary as input for running the child code (shown by the end of the arrow). Two-way arrows indicate an interaction, outlining a feedback relationship. Initially, `create_dataset.py` outputs the dataset containing the added vulnerability attribute, not initially present in the original dataset. This expanded dataset is then used by `train_model.py` to train the toxicity detection NN which is provided as input to `evaluate_fairness.py`, that establishes whether the model is fair, through the utilization of fairness metrics described in D3.1. These results are then provided as input to the visual user interface created by `visual.py`. The interactions with `modelscan.py` guarantee that no malicious code exists in the saved models.

Similarly, `train_private_model.py`, uses the methods described in Section 2.3.3 of D5.1 to train a privacy preserving model to be used by the Visual Component, and its results being visualized by `draw_plots.py`.

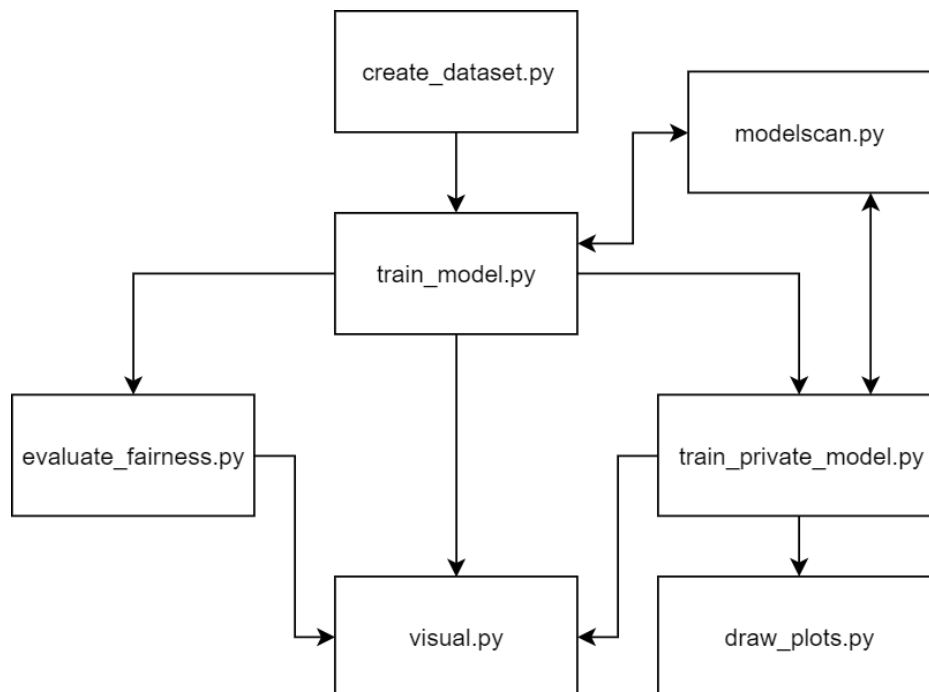


Figure 1: Common code execution order

Each file can be run after installing the required packages via pasting the following to a terminal:

```
python file_to_run.py
```

An exception to this is modelscan, whose execution is described in Section 6.

Most of the code requires no user interaction at all, since everything is automated, from creating the dataset to training the model, however many parameters are freely modifiable by the user, such as the learning rate of the training procedures, the amount of noise included in training the private model or the distributions existing in the dataset.

When running `visual.py` the user is greeted by the GUI described in Section 7 of this document, as well as Section 3.4 of D5.1. This GUI includes all the tools developed for D5.1 and D5.2, and provides a concise way to visualize their results, as well as additional features developed specifically for the Visual Component, as described in detail in D5.1.

The operation and description of each of the files outlined in Figure 1, are available in the following Sections of this document.

3 Demonstration

As mentioned above, the Visual Component consists of a demonstrator of the outcomes of the tools developed in the context of task T5.1. So far, (i) a toxic word highlighting mechanism to test whether the AI Fairness tool correctly differentiates a toxic post from a non-toxic one, and (ii) a mechanism to visualize metrics relevant to the PPML tool to assess its effectiveness, are incorporated into the Visual Component. A mechanism to evaluate the AI Cybersecurity tool is going to be integrated in the 2nd version of the tool.

A demo of the visual component is provided in this Section. As depicted in Figure 2 the maintainer can load posts from the platform by clicking on the “**Load Posts**” button, from the button list in **Feature A** (Figure 2, left). By clicking this button, a pop-up window will appear, where the technician will be requested to input a PATH (location on the computer) value leading to the files containing posts. Then, the posts will appear as a sorted list in the **Feature B** (Figure 2, right) and are selectable.

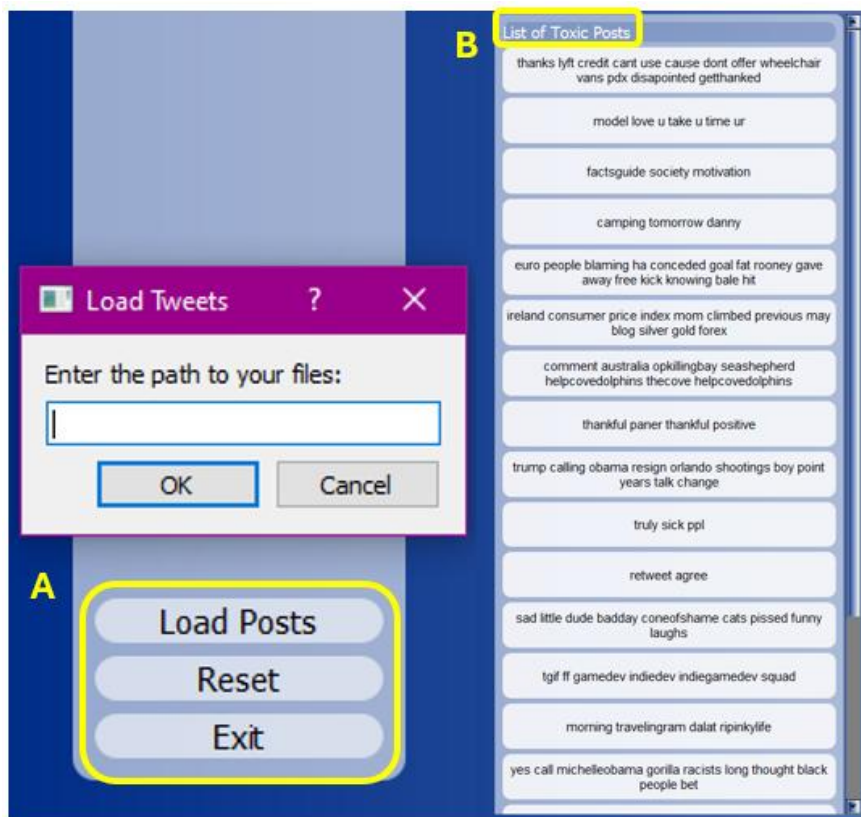


Figure 2: Loading and browsing platform posts on the Visual Component.

Upon selection, the posts appear in a text box area (**Feature C**) and then, words heavily associated with toxic posts by the AI Fairness tool are highlighted with a colour from a palette from green to red indicating their level of toxicity (i.e. words closer to red are toxic, while words closer to green are non-toxic). It is worth mentioning that in Figure 3, the word “motivation” is deemed as potentially non-toxic, while the word “society” is flagged as potentially slightly toxic. This happens because the toxicity detection model does not decide whether or not a specific excerpt is toxic solely based on the words that exist within it, but by establishing complex interactions between them, i.e. the meaning of the sentence. By contrast the word-toxicity-rate employed to highlight potentially toxic words, is extracted by post-processing the results of the neural network, in order to associate each word with its rate of appearance in posts deemed toxic which does not take into account the potentially complex relationships between the words as well as their respective positions and emphasis in the sentence.



Figure 3: Editing selected post from list.

In the following figure, Figure 4, an example of a toxic post, i.e. containing words that are deemed toxic, thus highlighted with dark orange color (Figure 4, left), and a non-toxic one with words highlighted in green (Figure 4, right), are depicted.

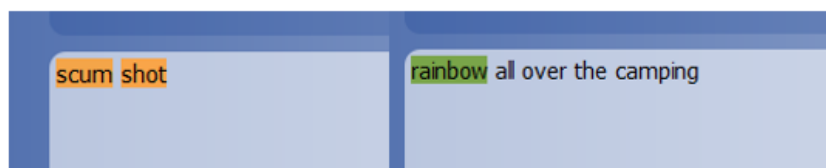


Figure 4: Example of potentially toxic words highlighting.

The maintainer/technician is also able to insert free text in the text box flagged as **Feature D** in Figure 5, which can be evaluated as toxic or non-toxic with the press of a button (**Feature E**).

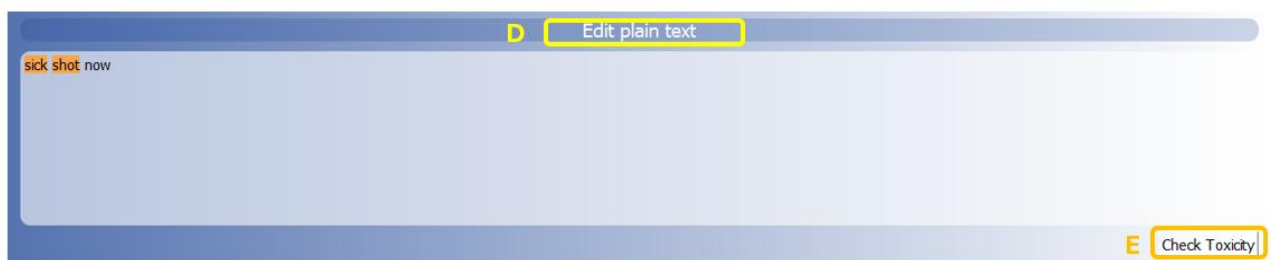


Figure 5: Detection of potentially toxic words on custom text written by user.

As for the association of the PPML tool with the Visual Component, the author of the selected post remains anonymous to maintain impartiality, fairness, and privacy, while the metrics for the threshold membership inference attack used to test the proposed Differential Privacy method are depicted as graphs in **Feature F**. The latter provides the engineer with useful information regarding the level of vulnerability of the underlying AI model to privacy attacks and is intended to be updated in a weekly or monthly basis in a subsequent version of the visual component. As shown in Figure 6, the metrics of *Area Under Curve (AUC)*, *Attacker Advantage*, and *Positive Predictive Value (PPV)* are visualized.

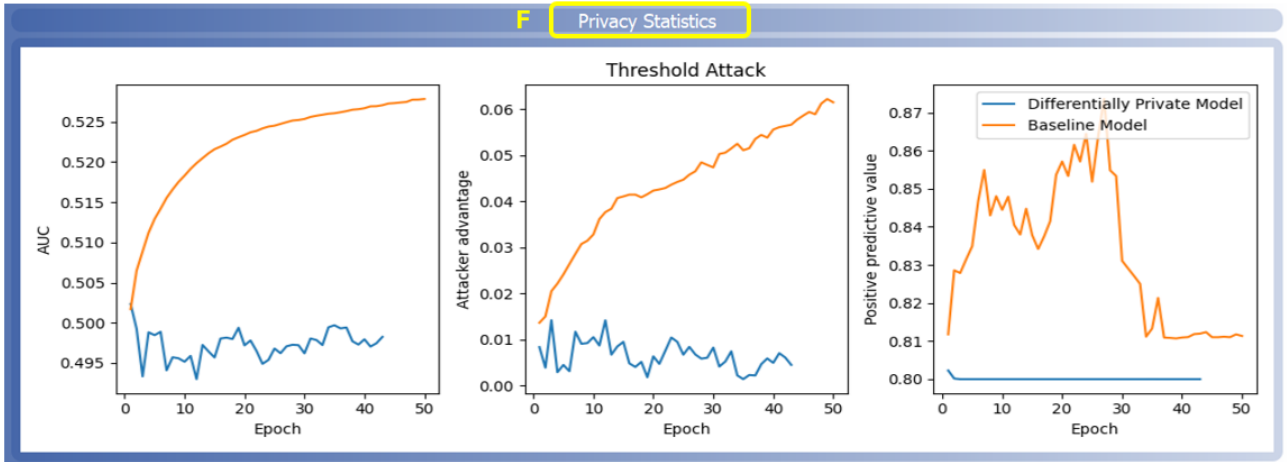


Figure 6: Privacy report based on model resilience against threshold attacks. Resilience is quantified via the metrics Area Under Curve (AUC), Attacker advantage and Positive predictive value, described in Section 2.3.2 of D5.1.

In Table 1, there is a short description of every element each Feature includes and its functionality.

Table 1 Visual component Features description.

Feature	Description
A	<p>“Load Posts” Button: By pressing this button a pop-up message appears requesting for a PATH value that leads to saved posts files from the platform</p> <p>“Reset” Button: It clears any text from sections C and D</p> <p>“Exit” Button: By pressing this button a pop-up message appears asking the user whether they would like to exit the application.</p>
B	A list of selectable posts. The posts appearing in the list stem from the Toxic Tweets Dataset (Iyer, 2021), used to test the functionality of our AI Fairness Tool.
C	A textbox area where posts selected from feature B appear. Words with slight or heavy toxic meaning are highlighted from green to red accordingly, while the user is able to edit these posts.
D	A textbox area where the user can input and process free text.
E	“Check Toxicity” Button: By pressing this button the text in Feature D is evaluated whether it contains toxic terms.
F	An area to plot the values of metrics associated with the threshold membership inference attack used to test the proposed Differential Privacy that our PPML tool employs.

Finally, an Overview of the Visual Component with all Features as described above is depicted in Figure 7.

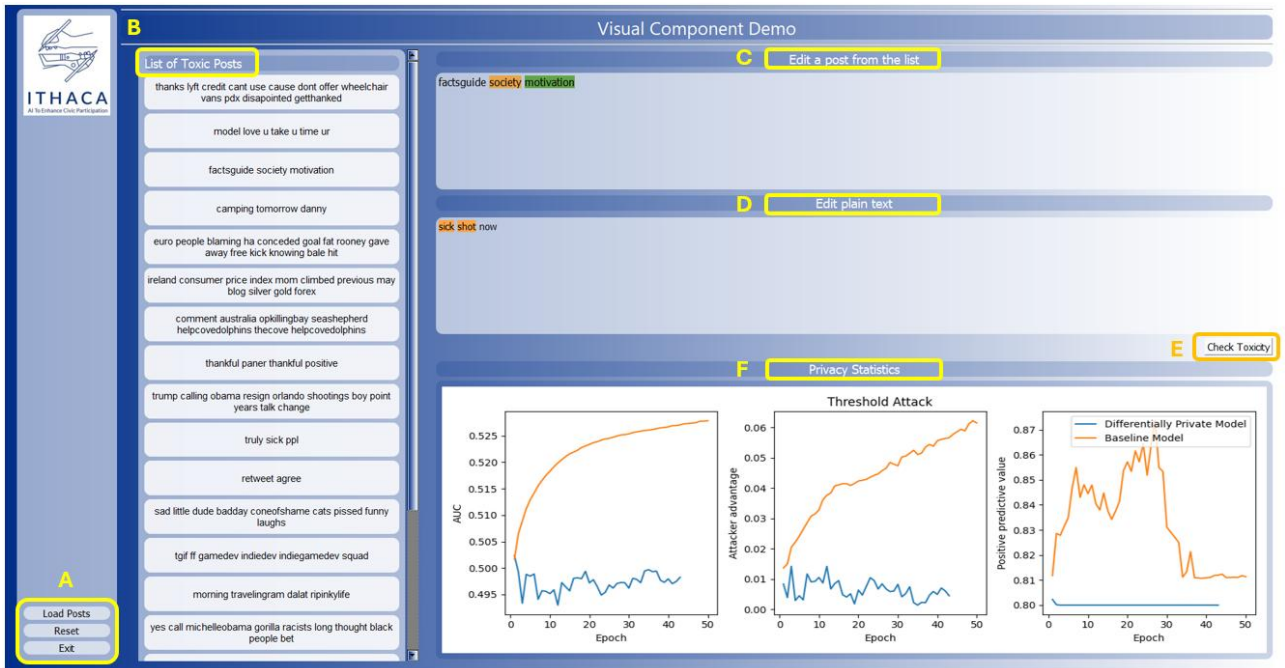


Figure 7: Full view of the Visual Component Tool.

4 AI Fairness Tool

In the context of T5.1, an AI Fairness tool was developed, to run alongside AI-based models of the ITHACA platform to assess the impartiality of the latter towards marginalized and vulnerable groups. The first version of this tool employs the metrics of *disparate impact*, *statistical parity difference*, *equal opportunity difference*, *between group generalized entropy error*, and *treatment equality* to evaluate a Natural Language Model (NLP) that detects the usage of toxic speech in short written excerpts. The values of these metrics (i.e. being over or under a certain threshold) indicate whether the NLP correlates discriminatory/offensive text with the group the author of the text belongs to, being biased towards vulnerable groups. For more details regarding the technical and legal requirements that led to the technical concept and initial development of the fairness tool please refer to Sections 2.2.1, 2.3.1, and 3.1 of D5.1.

The pipeline of the fairness tool, i.e. application of fairness metrics on an AI model, is done via a set of functions organized in a Python script. Below we provide the documentation of these functions, including functions for data loading and preprocessing as well as model loading.

Initially, the dataset is created via the following functions in the `create_dataset.py` source file:

load_dataset(file_path)

Load the dataset from a CSV file.

Args:

file_path: str, the path to the CSV file.

Returns:

df: DataFrame, the loaded dataset.

assign_vulnerability(df, prob_vul=0.2, prob_unfair=0.7)

Assign vulnerability status to rows in the dataset based on given probabilities.

Args:

df: DataFrame, the dataset to modify.

prob_vul: float, the percentage of the population that is vulnerable.

prob_unfair: float, the percentage of the vulnerable population that is unfairly treated.

Returns:

df: DataFrame, the dataset with assigned vulnerability statuses.

print_vulnerability_stats(df)

Print statistics about the vulnerability distribution in the dataset.

Args:

df: DataFrame, the dataset to analyze.

Returns:

None

save_dataset(df, file_path)

Save the DataFrame to a CSV file.

Args:

df: DataFrame, the dataset to save.

file_path: str, the path to save the CSV file.

Returns:

None

Example usage of functions mentioned above:

```
df = load_dataset("data/FinalBalancedDataset.csv")
df = assign_vulnerability(df)
print_vulnerability_stats(df)
save_dataset(df, "data/synthetic_fairness_dataset.csv")
```

Subsequently, the preprocessing procedure, and the model training, employ the functions described below. The usage of these functions can be seen in the `train_model.py` source file.

load_model(*checkpoint_directory*)

Loads a trained model from a checkpoint directory.

Args:

`checkpoint_directory` (str): The directory path where the model checkpoints are stored.

Returns:

model: The loaded model with weights restored from the best checkpoint.

Example:

```
>>> model = load_model("./path/to/model/")
```

preprocess_data(*data_df*)

Preprocesses the data by shuffling the dataframe, removing URLs from tweets, and preparing the input and target arrays for training.

Args:

`data_df` (pandas.DataFrame): The input data dataframe.

Returns:

tuple: A tuple containing the preprocessed input array (X) and target array (y).

Example:

```
>>> data = pd.read_csv("path/to/dataset")
>>> X, y = preprocess_data(data)
```

split_data(*X, y*)

Split the data into training and validation sets.

Args:

`X` (array-like): The input features.

`y` (array-like): The target labels.

Returns:

tuple: A tuple containing the training sentences, validation sentences, training labels plus vulnerability, and validation labels plus vulnerability.

Example:

```
>>> data = pd.read_csv("path/to/dataset")
>>> X, y = preprocess_data(data)
>>> train_sentences, val_sentences, train_labels_plus_vul, val_labels_plus_vul =
split_data(X, y)
>>> print("Training sentences:", train_sentences)
>>> print("Validation sentences:", val_sentences)
>>> print("Training labels plus vulnerability:", train_labels_plus_vul)
```

```
>>> print("Validation labels plus vulnerability:", val_labels_plus_vul)
```

make_predictions(*model*, *val_sentences*)

Makes predictions using the given model on the provided validation sentences.

Args:

model: The trained model used for making predictions.

val_sentences: The validation sentences to make predictions on.

Returns:

y_pred: The predicted labels for the validation sentences.

Example:

```
>>> model = load_model(checkpoint_dir)
>>> val_sentences = ["This is a positive sentence.", "This is a toxic sentence."]
>>> y_pred = make_predictions(model, val_sentences)
>>> print(y_pred)
```

evaluate_fairness(*y_pred*, *val_labels_plus_vul*)

Evaluate the fairness of a binary classification model.

Args:

y_pred (numpy.ndarray): Predicted labels of the model.

val_labels_plus_vul (numpy.ndarray): Validation labels with vulnerability information.

Returns:

None

Example:

```
>>> y_pred = np.array([0, 1, 0, 1, 0])
>>> val_labels_plus_vul = np.array([[0, 0], [1, 1], [0, 1], [1, 0], [0, 1]])
>>> evaluate_fairness(y_pred, val_labels_plus_vul)
```

print_fairness_metrics(*metric, val_labels_df, predictions_plus_vul_df*)

Prints fairness metrics and interpretations based on the provided metric object, validation labels dataframe, and predictions plus vulnerability dataframe.

Args:

metric: The fairness metric object used to calculate fairness metrics.
val_labels_df: The validation labels dataframe.
predictions_plus_vul_df: The predictions plus vulnerability dataframe.

Returns:

None

Example:

```
>>> val_labels_df = pd.DataFrame(
    {"toxicity": [0, 1, 0, 1, 0],
     "vulnerability": [0, 1, 1, 0, 1]})
>>> predictions_plus_vul_df = pd.DataFrame(
    {"toxicity": [0, 1, 0, 1, 0],
     "vulnerability": [0, 1, 1, 0, 1]})
>>> metric = ClassificationMetric(
    val_labels_df,
    predictions_plus_vul_df,
    privileged_groups=[{"vulnerability": 0}],
    unprivileged_groups=[{"vulnerability": 1}])
>>> print_fairness_metrics(metric, val_labels_df, predictions_plus_vul_df)
```

main()

Main function of the `evaluate_fairness` script.

Loads a model from a checkpoint directory, reads a synthetic fairness dataset, preprocesses the data, splits it into validation data, makes predictions using the model, and evaluates the fairness of the predictions.

Args:

None

Returns:

None

5 Privacy Preserving Machine Learning Tool

To maintain the privacy of data that AI models of the ITHACA platform would process, a Privacy Preserving Machine Learning Tool (PPML) was developed. In the first version of this tool, a Differential Privacy (DP) method that injects random noise into data to conceal any identifiable information in case of a potential attack, is implemented. The effectiveness of the PPML tool is assessed in the scenario of a *Membership Inference Threshold Attack* on the NLP for toxicity detection, via the metrics *Area Under Curve (AUC)*, *Positive Predictive Value (PPV)*, and *Attacker Advantage*, whose values indicate the success or failure of such an attack as discussed in Sections 2.3.2 and 3.2 in D5.1. Moreover, the legal requirements and quantitative criteria bounding the implementation of the PPML tool are discussed in Sections 2.2.2 and 2.3.2 of D5.1.

All the functions concerning the application of the privacy tool are outlined below:

`create_dp_model(train_sentences=None)`:

This Python function, `create_dp_model(train_sentences=None)`, is used to create a differentially private (DP) model for binary text classification tasks. It uses TensorFlow and TensorFlow Privacy libraries. The function is divided into several sections:

- *Text Vectorization*: The function creates a TextVectorization layer from TensorFlow's Keras API. This layer transforms strings into a list of integer indices representing words. The parameters for this layer include the maximum vocabulary length, the standardization method, the token split method, the output mode, and the output sequence length.
- *Vectorizer Training/Loading*: If `train_sentences` are provided, the function fits the TextVectorization layer to the training data and saves the vectorizer's configuration and weights to disk. If `train_sentences` are not provided, the function loads the vectorizer's configuration and weights from the disk.
- *Embedding Layer*: The function creates an Embedding layer, which transforms integer indices into dense vectors of fixed size. The parameters for this layer include the input dimension, the output dimension, the embeddings initializer, the input length, and the layer name.
- *Model Architecture*: The function defines the architecture of the DP model. The model takes a string input, applies the TextVectorization layer, applies the Embedding layer, applies a GlobalAveragePooling1D layer, and finally applies a Dense layer with a sigmoid activation function.
- *Differential Privacy Optimizer*: The function creates a DPKerasAdamOptimizer from the TensorFlow Privacy library. This optimizer implements the Adam algorithm with differential privacy. The parameters for this optimizer include the L2 norm clip, the noise multiplier, the number of microbatches, and the learning rate.
- *Model Compilation*: The function compiles the model with the binary cross-entropy loss function, the DP optimizer, and accuracy as the metric.

Args:

`train_sentences` (list, optional): A list of sentences used for training the TextVectorization layer. Defaults to None.

Returns:

`tf.keras.Model`: The compiled differentially private model for text classification.

class PrivacyMetricsCallback(tf.keras.callbacks.Callback)

The *PrivacyMetricsCallback* class is a callback used for computing privacy metrics during training. It takes several arguments in its constructor, including *epochs_per_report*, *x_train*, *x_test*, *y_train_indices*, *y_test_indices*, *batch_size*, and *model_name*. These arguments are used to initialize the attributes of the class.

Args:

- epochs_per_report* (int): Number of epochs between privacy reports.
- x_train* (numpy.ndarray): Training data.
- x_test* (numpy.ndarray): Test data.
- y_train_indices* (numpy.ndarray): Training labels.
- y_test_indices* (numpy.ndarray): Test labels.
- batch_size* (int): Batch size for predictions.
- model_name* (str): Name of the model.

Attributes:

- x_train* (numpy.ndarray): Training data.
- x_test* (numpy.ndarray): Test data.
- y_train_indices* (numpy.ndarray): Training labels.
- y_test_indices* (numpy.ndarray): Test labels.
- batch_size* (int): Batch size for predictions.
- epochs_per_report* (int): Number of epochs between privacy reports.
- model_name* (str): Name of the model.
- attack_results* (list): List to store the results of privacy attacks.

The *PrivacyMetricsCallback* class has a method called *on_epoch_end* which is called at the end of each epoch during training. Inside this method, there is a check to determine if the current epoch is a multiple of *epochs_per_report*. If it is, a privacy report is generated. To generate the privacy report, the method first uses the *model* attribute (which is assumed to be an instance of a TensorFlow Keras model) to make predictions on the training and test data (*x_train* and *x_test*). The predictions are stored in *prob_train* and *prob_test* respectively.

Next, the method creates an instance of the *PrivacyReportMetadata* class, which is used to store metadata about the evaluated model. The metadata includes the training and test accuracy, the epoch number, and the model variant label (which is set to the *model_name* argument).

The method then calls the *run_attacks* function, passing in the necessary inputs such as the attack input data, the slicing specification, the attack types, and the privacy report metadata. The *run_attacks* function is responsible for running membership inference attacks on the model.

The results of the attacks are stored in the *attack_results* attribute of the *PrivacyMetricsCallback* class. These results can be accessed and analyzed after training is complete.

Overall, the *PrivacyMetricsCallback* class provides a convenient way to compute privacy metrics during training by generating privacy reports and running membership inference attacks on the model.

train_private_model.py

A Python script that includes various function calls and operations related to training a machine learning model with privacy guarantees.

The first part of the script shows the training process using the *fit()* function of a *model_1* object created by calling the *create_dp_model* function. The *fit()* function takes in training data (*train_sentences* and *train_labels*), as well as other parameters such as the number of epochs, validation data, callbacks, batch size, and shuffle flag. This function trains the model and returns a history object that contains information about the training process.

Next, this script retrieves the latest checkpoint of the trained model using the *tf.train.latest_checkpoint()* function. The *checkpoint_dir* parameter specifies the directory where the model checkpoints are saved. The latest checkpoint is loaded into the *model_1* object using the *load_weights()* function.

After loading the best model, the script evaluates the model's performance on the validation data using the *evaluate()* function of *model_1*. The evaluation results are stored in the *model_1_evaluate* variable.

The code then calls the *compute_dp_sgd_privacy.compute_dp_sgd_privacy_statement()* function to compute and print a privacy statement. This function takes in various parameters such as the number of training examples, number of epochs, batch size, noise multiplier, and delta. It calculates the privacy guarantee provided by the differentially private stochastic gradient descent (DP-SGD) algorithm used for training the model.

The script creates an *AttackResultsCollection* object called *results* to store the attack results. The *AttackResultsCollection* class is a collection of *AttackResults* objects, which store the results of privacy attacks on the trained model.

A list of privacy metrics (*privacy_metrics*) that will be used for plotting the privacy vulnerabilities, is defined. Also, the *plot_by_epochs()* function is called to generate a plot showing the privacy vulnerabilities over the epochs of training. The plot includes multiple subplots, each representing a different privacy metric.

The script checks if a directory named *./results/dp_model/* exists. If it doesn't exist, the code creates the directory using the *os.makedirs()* function. If the directory already exists, it is removed using *shutil.rmtree()* and then recreated.

Moreover, the script saves the results object to the *./results/dp_model/* directory using the *save()* method of the *AttackResultsCollection* class. The *save()* method serializes the object and saves it to a pickle file.

The code concludes by calling *plt.show()* to display the generated plot of accuracy and loss over the epochs.

6 AI Cybersecurity Tool

Safeguarding the AI models of the ITHACA platform is crucial for maintaining trust in civic participation. For this, the ITHACA platform utilizes a cybersecurity tool, named **ModelScan** (<https://modelscan.ai>), to further protect AI models against adversaries. ModelScan is thoroughly described in Section 3.3 of deliverable D5.1, while the qualitative and quantitative criteria defining its non-functional and functional requirements respectively are analysed in Sections 2.2.3 and 2.3.3 of D5.1.

ModelScan scans AI models' files and evaluates their safeness by categorizing any unsafe embedded operation into four levels of severity: CRITICAL, HIGH, MEDIUM, and LOW.

To highlight the importance of scanning ML models, we will showcase two TensorFlow models. Initially, a model named '*SafeModel*', was created, which consists of convolutional and pooling layers, followed by a dense layer. This model is constructed to handle the image classification task of the fashion-mnist (https://www.tensorflow.org/datasets/catalog/fashion_mnist) dataset without embedding any risky operations or external file interactions. The second model follows a similar architecture but does include a risky operator. Both models' architecture is presented in the following Figure 8.

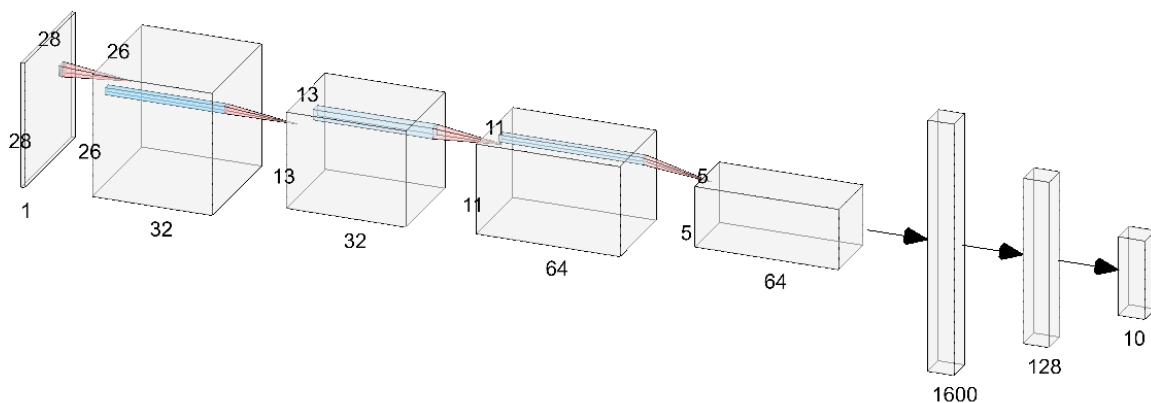


Figure 8: The models' architecture upon which ModelScan was tested.

class SafeModel (tf.keras.Model):

Args:

conv1 (tf.keras.layers.Conv2D): A 2D convolutional layer that takes as input a 28x28x1 image and applies 32 convolution filters, each of size 3x3.

pool1 (tf.keras.layers.MaxPooling2D): A 2D pooling layer that downsamples the feature maps by taking the maximum value over a window of size 2x2.

conv2 (tf.keras.layers.Conv2D): Similar to conv1, but with 64 filters.

pool2 (tf.keras.layers.MaxPooling2D): Similar to pool1 but with 2x2 window size.

flatten (tf.keras.layers.Flatten): This layer flattens the output in a single feature vector. It basically transforms that 2D matrix into a 1D.

d1 (tf.keras.layers.Dense): Densely connects the flattened feature vector with 128 neurons.

d2 (tf.keras.layers.Dense): The final layer, similar to d1 but with 10 neurons, one of each class for the classification task.

```

class SafeModel(tf.keras.Model):
    def __init__(self):
        super(SafeModel, self).__init__()
        self.conv1 = Conv2D(32, (3, 3), input_shape=(28, 28, 1))
        self.pool1 = MaxPooling2D((2, 2))
        self.conv2 = Conv2D(64, (3, 3))
        self.pool2 = MaxPooling2D((2, 2))
        self.flatten = Flatten()
        self.d1 = Dense(128)
        self.d2 = Dense(10, activation='softmax')

    def call(self, x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.flatten(x)
        x = self.d1(x)
        return self.d2(x)

safe = SafeModel()
safe.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
safe.fit(train_images, train_labels, epochs=1)
safe.save('safemodel')

```

Upon saving the model, we run ModelScan against it. ModelScan performs a thorough analysis by scanning various components of the saved TensorFlow model.

```
modelscan -p ./safemodel
```

The scan results for SafeModel show no issues, which indicates that the model is free of embedded unsafe code and is safe for deployment.

```

Scanning /content/safemodel/fingerprint.pb using
modelscan.scanners.SavedModelTensorflowOpScan model scan
Scanning /content/safemodel/keras_metadata.pb using
modelscan.scanners.SavedModelLambdaDetectScan model scan
Scanning /content/safemodel/saved_model.pb using
modelscan.scanners.SavedModelTensorflowOpScan model scan

```

```
--- Summary ---
```

```
No issues found! 🐼
```

```
--- Skipped ---
```

Next, another model named *'IOModel'*, was constructed. This model has a similar architecture to *SafeModel* but includes a risky operation: it attempts to read a file (*top_secret.txt*) during its execution. This kind of operation can be dangerous as it may lead to unauthorized data access if deployed in a production environment.

class IOModel (tf.keras.Model):

Args:

conv1 (tf.keras.layers.Conv2D): A 2D convolutional layer that takes as input a 28x28x1 image and applies 32 convolution filters, each of size 3x3.

pool1 (tf.keras.layers.MaxPooling2D): A 2D pooling layer that downsamples the feature maps by taking the maximum value over a window of size 2x2.

conv2 (tf.keras.layers.Conv2D): Similar to conv1, but with 64 filters.

pool2 (tf.keras.layers.MaxPooling2D): Similar to pool1 but with 2x2 window size.

flatten (tf.keras.layers.Flatten): This layer flattens the output in a single feature vector. It basically transforms that 2D matrix into a 1D.

d1 (tf.keras.layers.Dense): Densely connects the flattened feature vector with 128 neurons.

d2 (tf.keras.layers.Dense): The final layer, similar to d1 but with 10 neurons, one of each class for the classification task.

class IOModel(tf.keras.Model):

```
def __init__(self):
    super(IOModel, self).__init__()
    self.conv1 = Conv2D(32, (3, 3), input_shape=(28, 28, 1))
    self.pool1 = MaxPooling2D((2, 2))
    self.conv2 = Conv2D(64, (3, 3))
    self.pool2 = MaxPooling2D((2, 2))
    self.flatten = Flatten()
    self.d1 = Dense(128)
    self.d2 = Dense(10, activation='softmax')
```

```
def call(self, x):
    f = tf.io.read_file('top_secret.txt')
    x = self.conv1(x)
    x = self.pool1(x)
    x = self.conv2(x)
    x = self.pool2(x)
    x = self.flatten(x)
    x = self.d1(x)
    return self.d2(x)
```

```
io = IOModel()
io.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
io.fit(train_images, train_labels, epochs=1)
io.save('iomodel')
```

When ModelScan is applied to *IOModel*, it identifies the use of the unsafe ReadFile operator from the TensorFlow module. By applying the following to a terminal, we can see that the model is flagged with a HIGH severity issue because it poses a significant security risk:

```
modelscan -p ./iomodel

Scanning /content/iomodel/fingerprint.pb using
modelscan.scanners.SavedModelTensorflowOpScan model scan
Scanning /content/iomodel/keras_metadata.pb using
modelscan.scanners.SavedModelLambdaDetectScan model scan
Scanning /content/iomodel/saved_model.pb using
modelscan.scanners.SavedModelTensorflowOpScan model scan

--- Summary ---

Total Issues: 1

Total Issues By Severity:

- LOW: 0
- MEDIUM: 0
- HIGH: 1
- CRITICAL: 0

--- Issues by Severity ---

--- HIGH ---

Unsafe operator found:
- Severity: HIGH
- Description: Use of unsafe operator 'ReadFile' from module 'Tensorflow'
- Source: /content/iomodel/saved_model.pb
```

The scan results for *IOModel* highlight the criticality of scanning machine learning models for unsafe operations before deployment. Those scans ensure that models do not compromise security or violate data protection. By embedding ModelScan into the development and deployment pipelines, organizations can better secure their AI applications against potential exploits that could leverage deserialization vulnerabilities.

7 Visual Component

The Visual Component is an interface for human oversight over the above tools, namely, AI Fairness, PPML, and AI Cybersecurity tools, thus acting as a demonstrator of their functionality. It is intended for a responsible party (i.e. platform maintainer/engineer or human moderator) to test the functionality of the tools and have a level of control over them and the events arising from them.

Both the front-end and the back end of this component were developed using Python's PyQt5 library (Riverbank Computing Limited, 1998). Below the documentation for the Python PyQt5 script that implements the elements of the visual component is provided.

visual.py

This PyQt5 script defines a class *Ui_MainWindow* which is used to set up the user interface for the main window of the application. PyQt5 is a set of Python bindings for Qt libraries that can be used to create cross-platform applications with a graphical user interface (GUI).

The functions defined in *visual.py* are the following:

setupUI (self, MainWindow)

The *setupUi* method is where the UI elements are created and configured. Below there is a brief overview of what the *setupUi* method does:

- It sets up the main window, including its size, font, and background color.
- It creates a central widget and a horizontal layout for the main window.
- It creates a group box (*verticalGroupBox_2*) with a vertical layout (*verticalLayout_3*).
- It adds a label (*label_5*) to the group box *verticalGroupBox_2*, which displays the ITHACA logo image (*Ithaca_logo.jpg*).
- It adds three buttons (*pushButton_3*, *pushButton_2*, *pushButton*) to the group box *verticalGroupBox_2*. Each button is connected to a different function (*load tweets*, *clear_selection*, and *close*, respectively) that will be executed when the button is clicked.
- It adds the group box (*verticalGroupBox_new*) to the horizontal layout of the central widget, in which a scroll area (*scrollArea*) widget is added. In this scroll area, selectable short written excerpts (e.g. posts from the platform) are displayed.
- It creates another vertical layout (*verticalLayout*) in the central widget, which contains a *textBrowser* and a *plainTextEdit* widget, which are text boxes where the technician may process plain text. In *textBrowser* appear the posts from the *scrollArea* that are selected.
- In the same vertical layout (*verticalLayout*), a button is added (*pushButton_5*), which calls the *highlight_plain_text* function.
- In *verticalLayout* a widget (*widget*) is added which contains images of the metrics used to evaluate the level of privacy of the PPML tool.

Args:

self: The current instance of the class *Ui_MainWindow*

MainWindow (QtWidgets.QMainWindow): A widget of type MainWindow

Returns:

None

Example:

```
>>> ui = Ui_MainWindow()
>>> ui.setupUi(MainWindow)
```

readTweets(*self*)

Reads tweets using the `returnTweets` function and return the result.

Args:

self: The current instance of the class `Ui_MainWindow`

Returns:

X (list): A list of tweets.

Example:

```
>>> tweets = self.readTweets()
```

clickme(*self*)

Updates the `textBrowser` widget of the visual component with highlighted text based on the button pressed. This function is called upon the selection of the short, written excerpts that are displayed in the `scrollArea` widget.

This method retrieves the text from selectable post, splits it into words, and then checks if each word is present in the `unique_words` array. If a word is found, it is highlighted with a background color based on its index in the `unique_words` array. The highlighted text is then displayed in the text browser.

Args:

self: The current instance of the class `Ui_MainWindow`

Returns:

None

Example:

Connect the function with the respective button, thus, while the button is pressed, the `clickme` function will be called.

```
>>> pushButton_32.clicked.connect(self.clickme)
```

highlight_plain_text(*self*)

Highlights words toxic words based on their toxicity level in the `plainTextEdit` element using HTML formatting.

This method takes the plain text from the `plainTextEdit` widget, splits it into words, and checks if each word is present in the `unique_words` array. If a word is found in the array, it is highlighted with a background color based on its index in the `unique_words` array.

The highlighted text is then displayed in the `plainTextEdit` widget.

Note: The `unique_words` array is loaded from the `"/data/unique_words.npy"` file.

Args:

self: The current instance of the class `Ui_MainWindow`

Returns:

None

Example:

```
>>> self.pushButton_5.clicked.connect(self.highlight_plain_text)
```

load_tweets(*self*):

Loads posts (e.g. tweets from a dataset) from a specified path.

This method prompts the user to enter the path to their files and then uses the path to load the preferred files (i.e. files containing short written excerpts, e.g. posts).

Args:

self: The current instance of the class *Ui_MainWindow*

Returns:

None

Example:

```
>>> self.pushButton_3.clicked.connect(self.load_tweets)
```

clear_selection(*self*)

Clears the content of the *textBrowser* and *plainTextEdit* text boxes.

Args:

self: The current instance of the class *Ui_MainWindow*

Returns:

None

Example:

```
>>> self.pushButton_2.clicked.connect(self.clear_selection)
```

close(*self*)

Closes the PyQt5 application. Upon a button press, a pop-up window that asks the technician whether they would like to close the application appears. The technician may either press a *yes* or a *no* button. The *yes* button will successfully close the application, while the *no* button will cancel the operation and the pop-up window will disappear.

Args:

self: The current instance of the class *Ui_MainWindow*

Returns:

None

Example:

```
>>> self.pushButton.clicked.connect(self.close)
```

8 References

R

Riverbank Computing Limited. (1998). PyQt5. [Online] Available at: <https://pypi.org/project/PyQt5/>. Accessed, 14(05), 2023.