Designing and building smart privacy assistants for improving data dashboards

Project-I (CS47007) report submitted to

Indian Institute of Technology Kharagpur

in partial fulfilment for the award of the degree of

Bachelor of Technology

in

Computer Science and Engineering

by

Shashvat Gupta

(19CS30042)

Under the supervision of

Professor Mainack Mondal

Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

Autumn Semester, 2022-23

November 7, 2022

DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(d) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: November 7, 2022 (Shashvat Gupta)
Place: Kharagpur (19CS30042)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

KHARAGPUR - 721302, INDIA

CERTIFICATE

This is to certify that the project report entitled "**Designing and building smart privacy assistants for improving data dashboards**" submitted by **Shashvat Gupta** (Roll No. 19CS30042) to Indian Institute of Technology Kharagpur towards partial fulfilment of requirements for the award of degree of Bachelor of Technology in Computer Science and Engineering is a record of bona fide work carried out by him under my supervision and guidance during Autumn Semester, 2022-23.

Date: November 7, 2022

Place: Kharagpur

Professor Mainack Mondal
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur - 721302, India

Abstract

Name of the student: **Shashvat Gupta** Roll No: **19CS30042**

Degree for which submitted: **Bachelor of Technology**

Department: **Department of Computer Science and Engineering**

Thesis title: **Designing and building smart privacy assistants for improving data dashboards**

Thesis supervisor: **Professor Mainack Mondal**

Month and year of thesis submission: **November 7, 2022**

In today's world, user activity collection is a common phenomenon. To keep this collection in check, companies have introduced data dashboards where one could view and delete their collected data. One such popular dashboard is the **Google Activity Dashboard**. However, the dashboard allows users to keep their activity data in check "only in theory". Studies conducted by Sharma and Mondal (2022) show that while the dashboard provides complete control to the user, it is highly unusable rendering its existence moot. The study goes on to suggest ways to improve the dashboard design using Machine Learning. The idea is to use an automated classifier to filter data as sensitive or not sensitive. In this study, we implement a **smart privacy assistant plugin** which renders a dashboard according to the study's suggestions. We also conduct a study to evaluate the usability of the new dashboard to collect more insights and further suggestions.

Acknowledgements

I would first like to thank my thesis advisor Prof. Mainack Mondal. The door to Prof. Mainack's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Shashvat Gupta

Contents

List of Figures

Abbreviations

GVA	**G**oogle **V**oice **A**ssistant
ML	**M**achine **L**earning
GAD	**G**oogle **A**ctivity **D**ashboard
JS	**J**ava**S**cript
DOM	**D**ocument **O**bject **M**odel
UI	**U**ser **I**nterface
*****	To be implemented

Chapter 1

Motivation

1.1 Introduction

Today, most corporations collect and analyze huge amounts of data associated with the activities of their users. This is done to provide a more immersive and personal experience to the users. However, this collected data may contain sensitive information (e.g., personal voice recordings) that users might not feel comfortable sharing with others and might cause significant privacy concerns. To curb such concerns, service providers like Google present their users with a personal **data dashboard** called the "**My Activity Dashboard**", which allows them to view and delete collected activity data.

Studies conducted in the paper Sharma and Mondal (2022) conclude inadequacy in the current design of the **Google Activity Dashboard**. The main issues are a culmination of the **huge amount of collected data** and the lack of any organisational creativity towards the data elements. This uncovers a need to make data dashboards more usable by helping users **uncover sensitive data elements**. The paper further discusses a possible solution by evaluating the **feasibility of using an ML model** to predict whether a particular data element will be perceived by the user as sensitive or not.

Our study consists of **implementing the changes** suggested in the paper Sharma and Mondal (2022), evaluating the **change in usability of privacy dashboards** and suggesting further issues, limitations and improvements over the resulting dashboard.

1.2 Problem Statement

Our study addresses the problems in Google Activity Dashboard highlighted by the study performed in the paper Sharma and Mondal (2022). Some of the issues highlighted by the paper are as follows:

- Lack of actionable knowledge regarding management of data collected by **GVA**
- Long-time users expressed **need of assistance** in using the Activity dashboard
- Dashboard design becomes very **unusable in real-life scenario** as amount of data piles
- Most participants were likely to delete some data upon recommending sensitive data elements
- Established a strong demand for a **recommender system** for sensitive data elements

These findings suggest the underlying importance of a high accuracy sensitive-element recommendation system to improve the utility of data dashboards. In our study, we develop a **chrome plugin** that would act as a **smart privacy assistant**. The job of the plugin is to:

1. **Scrape** data elements from Google Activity Dashboard
2. Use ML model to predict the **probability of a data element being sensitive**
3. Render a more data-driven **privacy dashboard** based on the prediction of the classification model

Further, we would like to evaluate the usability of this plugin and state improvements.

Chapter 2

Software Components and Workflow

2.1 Components

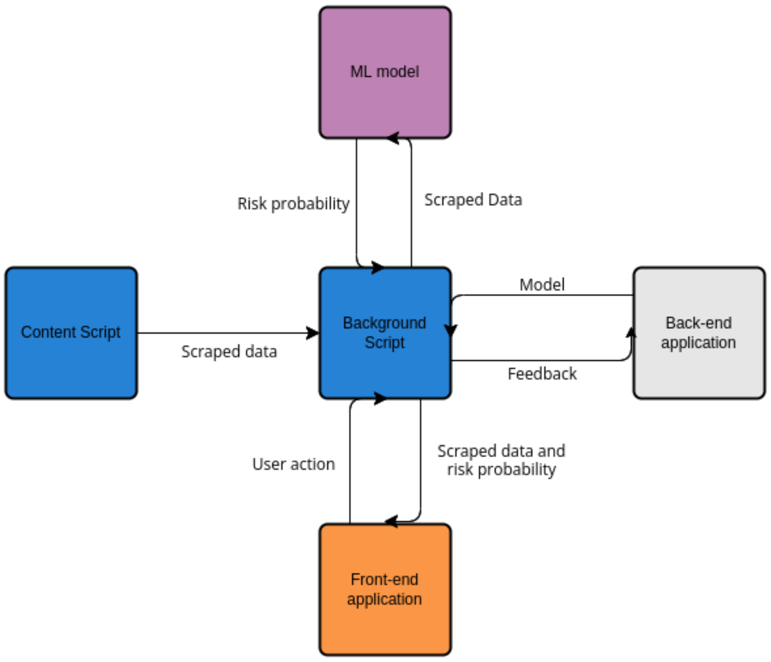


Figure 2.1: Data flow between components

2.1.1 Content Script

Content scripts are files that run in the context of web pages. By using the standard **Document Object Model** (DOM), they are able to read details of the web pages the browser visits, make changes to them, and pass information to their parent extension. Here, the Content Scripts run whenever the user visits the activity page and performs the following actions:

1. Puts an **overlay** over the page to mask the scraping activities
2. Starts a **scraper** to scrape data from Google Activity Dashboard
3. Removes the overlay when the scraping job is finished
4. Sends the collected data to plugin's **background script**

Scraper Library

This library was developed by us for the sole purpose to scrape all relevant data elements from the Google Activity Dashboard page. The scraper from the library is called from the content script as the library relies on access to the page's **DOM** to collect the data elements. The library consists of class definitions and implementations for the following:

- Classes definitions for collected data
- Methods to run ML prediction model on a data element
- Methods to render a dashboard component for a data element
- Dashboard component definition and styling for each data type
- Implementation of Scraper class to scrape Google Activity Dashboard

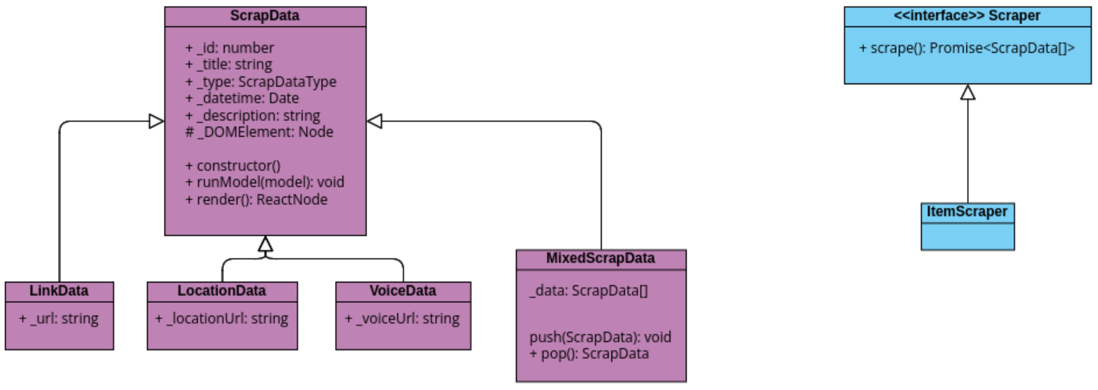


FIGURE 2.2: Class diagram for Scraperlib

Name	Purpose
ScrapData	It is the base class which is inherited by all data containing classes of the library. It stores basic information about a data instance.
LinkData	It is a data containing class that inherits the ScraperData class. It stores links in a data element.
VoiceData	It is a data containing class that inherits the ScraperData class. It stores voice recordings in a data element.
LocationData	It is a data containing class that inherits the ScraperData class. It stores location coordinates in a data element.
MixedScrapData	It is the most important data containing class. It maintains a list of ScrapData object instances and hence is used to store data elements that contain more than one type of data types.
ItemScraper	It is a scraper class which is used to scrap data from Google activity dashboard. The class implements various methods to scrape data elements in varying fashions. The ’**scrap**’ method returns a list of **MixedScrapData** type objects.

TABLE 2.1: Class definitions

Working of the Scraper

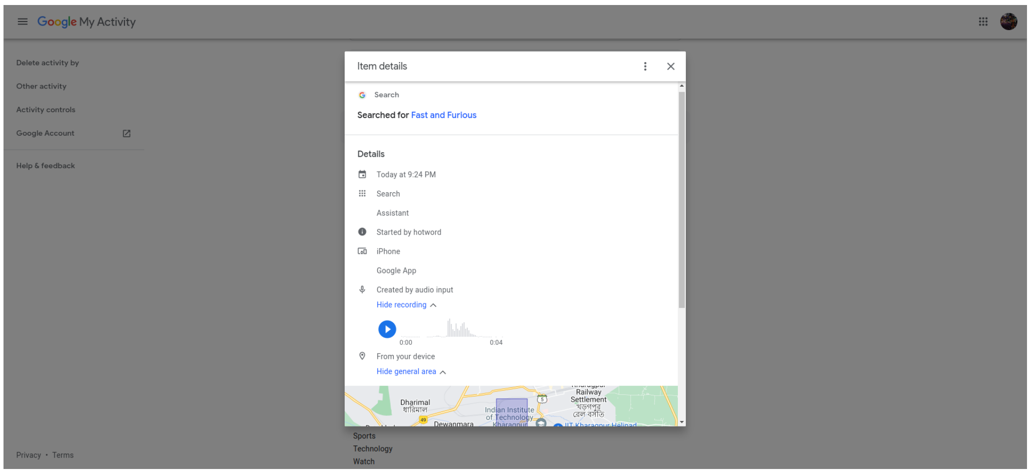


FIGURE 2.3: GAD Dialogue for a data item

For each data item, the dashboard provides a detailed dialogue box. This dialogue box contains all necessary information to be scraped. The scraper bot first opens the dialogue box of a data element. From this dialogue box, the scraper bot scans nodes with particular information type e.g. hyperlinks, voice recordings, location data etc. This information is scrapped and stored in an object instance of the respective type. All different types of objects that are scraped from a single data element are packaged together in a **Mixed-ScrapData** object. After this scraping is complete, the scraper bot closes the dialogue box and continues on the next element until all data is scraped. Finally, the scraper bot returns a list of MixedScrapData objects, each object corresponding to a data element in the dashboard.

Approximating time consumed: Upon implementation and testing, we found out that the most time taking step out of the algorithm described above is the **opening and closing of the dialogue box**. A qualitative explanation for the fact being that opening and closing of a dialogue box renders a new component and brings changes to the HTML DOM, while traversing and scraping of nodes is faster as it only reads DOM elements and no changes to the DOM are introduced ($T(write\_to\_DOM) >> T(read\_from\_DOM)$). Approximating time to open and close a dialogue box be $3s$ each for a 100 data elements (usual count in data dashboard), it would take $10m$ to scrape all data elements during which the overlay would hog the entire page. This is a very huge wait and would most likely lead to users not using the plugin.

We overcame this challenge by opening all dialogue boxes together, scraping data from a box, closing it on the spot and moving to the next dialogue box. This cuts the time substantially to $40s$. This is because now the program does not wait for a particular dialogue box to open, instead opening all dialogue boxes together only takes $20s$ due to heavy computational traffic. After all dialogue boxes are open, the programs scraps for data and closes the dialogue. While the dialogue is getting closed, the program starts scraping data from the next dialogue itself, leading to close time getting utilized. A waiting time of less than $1m$ is acceptable from user point of view.

2.1.2 Background Script

Extensions are an **event-based system**. Extensions monitor events in their background script, then react with specified instructions. A background script is loaded when it is needed, and unloaded when it goes idle.

Event	Handler Description
Every time Script is loaded	ML model stored with the extension is loaded into an instance of ModelAPI class expained in 2.1.3
On Installation	Open the Front-end application in a seperate tab
Clicking extension's icon	Open the Front-end application with scraped data stored in extension's local storage and send message to content script to rescrape the data from dashboard
Message from content script	Predict risk probability for received data using model loaded in ModelAPI instance and store the data (with probability) to extension's local storage
Weekly alarm*	Send the collected feedback data from user to back-end application for continual retraining of ML model, receive and load the new model in response
User action in front-end*	Map and propagate user actions in front-end (like deleting a data element) to Google Activity Dashboard

TABLE 2.2: Events handled in background script

2.1.3 ML model

According to findings of the paper Sharma and Mondal (2022), **XGBoost** ML model introduced in Chen and Guestrin (2016), is found to perform the best to classify data as sensitive or not. As per the suggestions of the paper, we implemented an XGBoost model with 90% accuracy.

A good trait of a recommendor system is to learn/personalize recommendations according to user's feedback. To incorporate the same, we decided to perform **continual learning** (Ke et al. (2021)) over the XGBoost model using the **user's feedback** as extra input. Continual Learning is a concept to learn a model for a large number of tasks sequentially without forgetting knowledge obtained from the preceding tasks. We conducted an experiment to show that continual learning improves the accuracy of a model by repetitively using continual training on a model and measuring its accuracy. The payoff on applying continual learning can solve the issue raised around the accuracy of the classification model in Sharma and Mondal (2022).

Loading and Running model in chrome extension

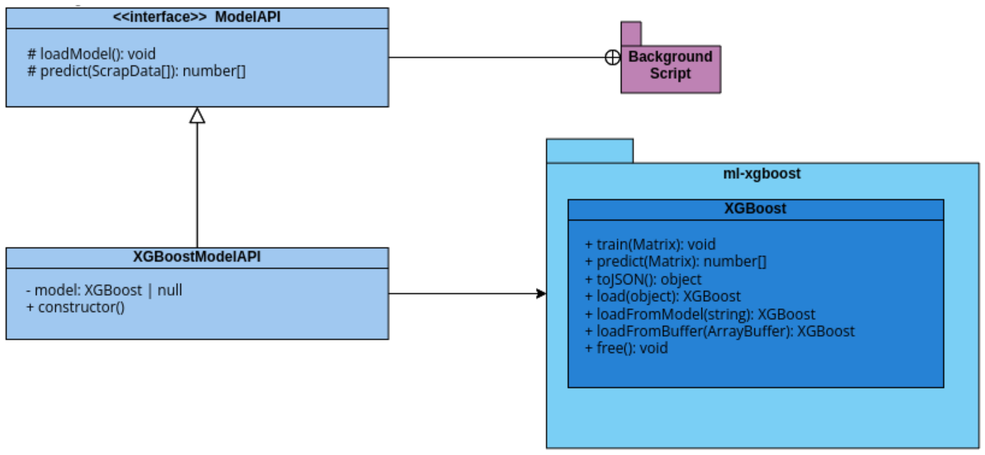


Figure 2.4: Class Diagram for ModelAPI

We introduced an interface called **ModelAPI** which is utilized by the chrome extension's background script. For a type of model (e.g. XGBoost), we develop a wrapper class implementing ModelAPI methods. In future, more number of experimental/custom models can be attached to the extension and even switched through the *Settings* menu. Currently, we have only implemented **XGBoostModelAPI** class that is a wrapper class for XGBoost model implementing the ModelAPI methods.

The **XGBoostModelAPI** class implements methods of **ModelAPI** which are *loadModel* and *predict*. This class acts as a **wrapper class** to use methods of **XGBoost** model in our extension's background script. In our extension, we kept the pre-trained and saved model in **JSON** format as '*model.json*'. To load and interpret this saved model file, we used the library **ml-xgboost**. The library was developed and merged into the main **xgboost** repository in response to an **issue** raised in the community regarding the usage of XGBoost models for prediction and training in the browser itself.

The library uses **WebAssembly**. WebAssembly is a **binary instruction format** and virtual machine that brings near-native performance to web browser applications, and allows developers to build high-speed web apps in the language of their choice. Here, the developer used the **C API** to load models saved from any other programming languages and generated a WebAssembly binary to use the program in browser. The library also provides implementation of a wrapper class (**XGBoost**) in JS over the the loaded model in C language for easy use in browser runtime. The library uses **ml-matrix** as a dependency for complex mathematical operations on matrices required while training and predicting using ML models. Hence, the input vector for prediction needs to be a **Matrix** defined in the ml-matrix library.

We faced a some major challenges in using the library **ml-xgboost** to load the XGBoost model JSON file:

1. Library exported a promise in **ES5** format but the browser is using **ES6** Javascript
2. No **Typescript** support defined (no types file) by the library

To solve the first issue, we cloned and implemented our version of the library changing the export and import to ES6 format, while keeping all other logic same. We also changed the export from a promise to components required to generate the same promise in browser. For the second issue, we developed typescript support for the library ourselves by writing a *types* file for all necessary classes of the library.

2.1.4 Front-end application

ReactJS is an amazing interface to develop interactive and complex front-end applications. One of the most significant advantages of React is that its DOM is declarative. In other words, the UI is adjusted when developers change the app's state when interacting with DOM. We try to develop a more **usable** and **data-driven** dashboard using the ReactJS framework.

The dashboard is rendered in a seperate tab whenever the extension's icon is pressed. The moment the dashboard is rendered, it fetches all the data stored by the background script (which includes the scraped data and their respective risk percentages) and stores it in the state of the dashboard. This stored information is rendered on a data-driven and usable UI. All user actions (like deleting an activity) on this UI are mapped to appropriate actions on the actual Activity Dashboard/Plugin State.

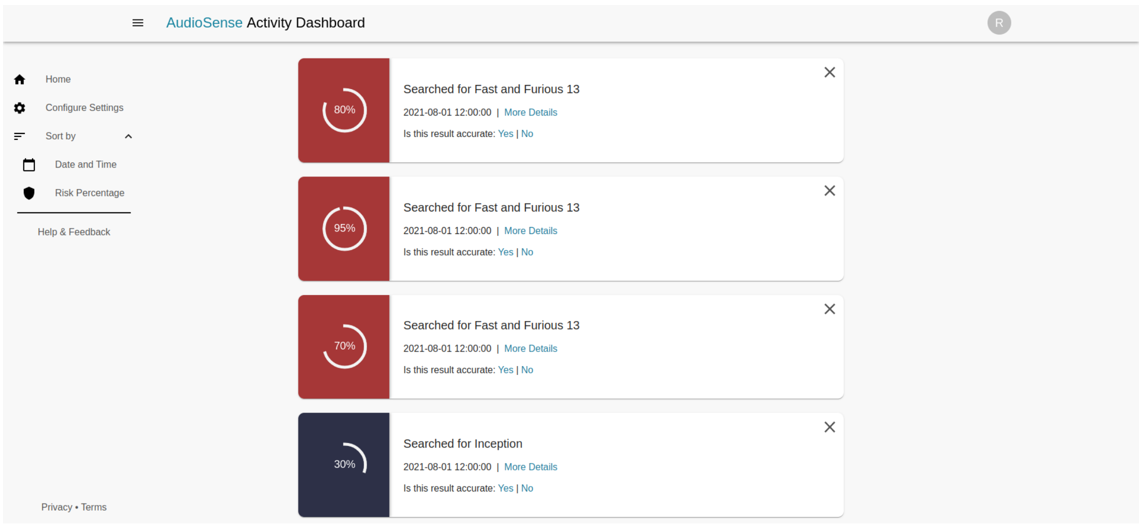
Features of the Interface

The interface is aimed to be more usable than the Google Activity Dashboard. In the least, it must include the most usable and important features provided by GAD. Some must-need features (implemented and required) are as follows:

- View all data elements in an ordered fashion
- View description of data elements in depth
- Take feedback regarding classification of data items
- Allow user to change the ordering fashion according to their convenience
- Nudge user about sensitive data items
- Allow user to delete an activity*
- Allow user to delete all sensitive-suggested elements*
- Provide interface to safely change settings of the plugin*
- Allow user to edit their personal information*
- View privacy and terms of the plugin*
- Provide a help guide for the plugin*

This list includes the basic features of the interface. After further study and analysis, more features might be added to the interface according to users' response (Section 3.2.1).

Interface Design



Searched for Fast and Furious 13
2021-08-01 12:00:00 | More Details
Is this result accurate: Yes | No

Searched for Fast and Furious 13
2021-08-01 12:00:00 | More Details
Is this result accurate: Yes | No

Searched for Fast and Furious 13
2021-08-01 12:00:00 | More Details
Is this result accurate: Yes | No

Searched for Inception
2021-08-01 12:00:00 | More Details
Is this result accurate: Yes | No

FIGURE 2.5: Plugin Dashboard

A simple prototype for the dashboard is designed on **Figma**. The *current* design of the interface follows the GAD interface while improving it with collected results and popular design principles.

Some design principles that are closely followed in the plugin interface:

- **Learnability**: We assume that most of the target-audience of the plugin has used Google Activity Dashboard and so, keeping our interface design close to GAD would transfer user's **learnability**, making it quicker for them to get accustomed to the plugin's basic functionalities.
- **Logical Layout (the Inverted Pyramid)**: The principle says that "*display the most significant insights on the top part of the dashboard, trends in the middle, and granular details in the bottom*". In our interface, all the sensitive data items are kept at the top of the interface. This cuts the user's trip to our dashboard much smaller as they get the focus points on a silver platter.
- **5 Second Rule**: The principle says that "*dashboard should provide the relevant information in about 5 seconds*". Numeric data sometimes might fail to provide *5 second* insights to the data but visual organisation in the form of colors never fails to do so. Our dashboard uses **"red" to denote sensitive elements** and **"blue" for the safe elements**. This also ties to **general human nature** which associates the color "red" with danger and "blue" with calm energies.
- **Minimalism**: The principle says that "*Less is more*". Our dashboard only shows the **minimum data** required by the user to make inferences at the first glance. Details can be opened for elements upon user's choice. This helps the user to make quick decisions by showing necessary data for all elements and detailed description only for elements queried by the user.
- **Tooltip**: Best use of tooltips is to explain a functionality (e.g. function of a specific button) to the user for once, after which the user can easily learn it and won't have to look up for the tooltip again i.e. **one-time explanation**. So, we used tooltips at places where we believed the user might need an explanation at most once e.g. use of cross button, significance of number inside pie chart etc.
- **Customisability**: The principle states that "*Everything suits the user*". We tried to provide as much customisability over the dashboard as possible by allowing user

to sort data elements according to their choice.

2.1.5 Back-end application

The work on this component is in progress. We decided to build the server using **Django** framework supported by Python. The back-end will have two endpoints, one endpoint will be used to train the model according to user information received via **POST** request from front-end application. The other endpoint is used to fine tune the model based on user's feedback data. In each end point, we send a compressed model as the response of the server.

2.2 Workflow

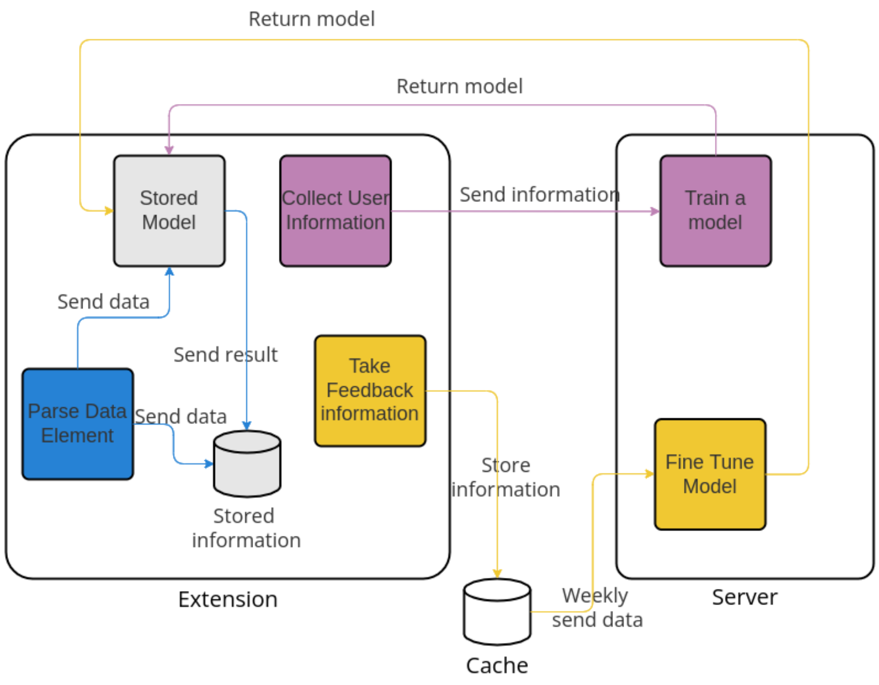


FIGURE 2.6: Broad workflow of system

2.2.1 Post Installation

This workflow is shown in **purple** in Figure 2.6.

1. The user is prompted to fill a form which asks for the user's personal information e.g. age, gender etc. This information is required to to train a good classification model. As stated in Sharma and Mondal (2022), **user features tremendously improve the model's accuracy**. The user may opt out of providing this information and in that case the model is trained using some default user features.
2. After the user fills the form, the information is sent to the back-end application by the extension's background script via a **POST** request.
3. The back-end uses this information to train a **XGBoost classification model** with premium accuracy. This model is saved, compressed and returned as response to the extension's background script.
4. On receiving the model, the background script stores the model. This model will be used to predict the **risk probability** of data elements while the extension is generally used.

2.2.2 General Usage

This workflow is shown in **blue** in Figure 2.6.

1. When generally used, the user is supposed to visit the **Google Activity Dashboard**. On visiting the dashboard, the plugin's content script scraps all the data elements and parses their features. During the scraping occurs, an **overlay requesting user to wait** would hog the dashboard page to cover all the scraping activities performed by the bot.
2. After content script parses all the data, it removes the overlay and sends the data to the background scripts which in turn send the data to the stored model. Using the stored model, the background script gets the **risk probability** of all the data elements. All of this information (scraped data and their respective risk probability), is stored in the extension's local storage. Note that this happens in the background.
3. To view the plugin's dashboard, the user is supposed to click the extension's icon in the extension bar of chrome browser. On clicking the icon, the plugin's front-end application (dashboard) is rendered in a new tab.

4. While rendering, the dashboard fetches the stored information through the background script and sets it as a state of the application. This data is then presented in a more usable and interactive manner.
5. Any **user action** is mapped to an action in the Activity dashboard, hence the plugin dashboard acts as a more usable wrapper of the Activity dashboard. However, the user action of providing feedback regarding results of the classification has a different and more complex flow.

A very important keynote in this workflow is that **no user data is being sent/leaked to any outer source**, hence, the user's data is **secure** from any third party intervention.

2.2.3 Providing Feedback

This workflow is shown in **yellow** in Figure 2.6

1. The user is supposed to provide feedback regarding the **authenticity of classification result** for a data element through the plugin's interface.
2. When the user provides a feedback, the feedback is stored in a simulated **cache** in the extension's local storage. This feedback data is stored for a week.
3. Each week, all this collected data (only the features) along with the stored model, is sent to the back-end application by the extension's background script. The back-end application **fine-tunes** the model (applies continual retraining) and sends the tuned model in response.
4. On receiving the model, the extension's background script replaces the old model with the new model. This way the model continually improves and personalises the classification according to the user's choices.

Chapter 3

Challenges and Future Work

3.1 Challenges

3.1.1 Time Optimisation

As discussed in 2.1.1, time of opening and closing of a dialogue box governs the time complexity of data scraping, since changing the DOM structure requires a lot more time than reading and parsing information from it. Here, we will perform a more quantitative analysis of the time complexity. Since, the time of scraping is much less than opening and closing of dialogue box, we will assume the **time of scraping** ≈ 0.

For Algorithm I, for each data element (100 data elements is default), we open the dialogue box, scrap the data and then close the dialogue box.

$$\begin{aligned} open\_time &= close\_time = 3s \\ Total\_time &= 100 * (open\_time + 0 + close\_time) = 600s = 10min \end{aligned} \tag{3.1}$$

For Algorithm II, we open all dialogue boxes together. Note that this the bot prompts each dialogue box to open sequentially only but it moves to the next prompt before the interface opens the previous dialogue box completely. After all dialogue boxes are open, the bot start scraping the boxes and closing it as soon as the scraping is done. Please note that here as well, the bot just prompts the interface to close the dialogue box and continues scraping the next box before the previous one is completely closed. In this algorithm, the Activity dashboard needs to sustain heavy computation due to quick and many requests to open and close dialogue boxes. However, experimentally we observe that:

$$\begin{aligned} open\_time\ of\ 1\ box &= 3s \\ open\_time\ of\ 100\ boxes\ if\ done\ separately\ &= 3 * 100s = 300s \\ open\_time\ of\ 100\ boxes\ done\ together &= 20s \end{aligned} \tag{3.2}$$

This optimisation occurs due to overlapping of opening times between dialogue boxes but if the overlap was perfect the time should have been $3s$. This implies that the bot takes $\approx 0.17s$ to move to the next dialogue box in a computation strained environment which explains the imperfect overlap.

$$\begin{aligned} open\_time\_100 &= close\_time\_100 = 20s \\ Total\_time\_100 &= open\_time\_100 + close\_time\_100 = 40s < 1min \end{aligned} \tag{3.3}$$

Thus, we trade computational load with time consumed for scraping.

3.1.2 Audio Scraping

As shown in the Figure 2.4, all necessary data is shown in the dialogue box including the voice recording. Now, scraping the audio proved to be a huge challenge as it cannot be scraped through DOM. The DOM for the voice component only had the link to a google storage link. Request from any bot has been blocked on that domain by Google due to security concerns.

As a workaround, we focused on **directly extracting the features from the audio** instead of downloading the audio first and then extracting its features. To extract audio from audio playing in the browser we used the library **Meyda**. The JS library provides vast variety of methods to extract different types of features from audios including real-time, offline etc. We use this library by attaching it's **real-time listener** to the DOM's audio node, **muting** the audio node and then **playing** it. From user point of view, this scraping will be invisible with the overlay since the audio is already muted. The listener captures the audio's buffer received by the node when it is playing and extracts the required features from it. This was a huge challenge given the inadequacy of information from DOM, lack of feature extraction libraries from audio in JS and importance of keeping this task invisible to the user.

3.1.3 ML model in browser

A huge and very vital challenge which aroused was to unpack and load the saved XGBoost model in the browser's environment. This task was challenging due to a lot of reasons.

1. The project hinged on the success of this task which increased its importance
2. Lack of libraries for unpacking or using ML models in browser environment
3. Existing libraries could only unpack basic ML models and not XGBoost model
4. Format in which model was saved was **human-unreadable**, forbidding us to write our own interpreter
5. After long search and finding the library **ml-xgboost**, the library was very much crude and had issues listed in the section 2.1.3.

We improved upon the library **ml-xgboost** as described in section 2.1.3 and used the library in our extension. Now, we can easily load, unpack and use our saved XGBoost model.

3.2 Future Work

3.2.1 Participatory Design

Participatory design (Velden and Mörtberg (2014)) is a democratic process for design (social and technological) of systems involving human work, based on the argument that users should be involved in designs they will be using, and that all stakeholders, including and especially users, have equal input into interaction design. The study focuses on constantly incorporating user's suggestions in the dashboard until, the design reaches a state where it is usable enough for anyone.

We will be performing a **Co-design or participatory design study** on the plugin's dashboard. This will make the interface more usable by directly including the needs of the people who will be using it in the future.

3.2.2 User feedback to improve model

We plan on having two kinds of continual training for our XGBoost model.

1. **Large stride** continual learning of each user's model to personalise it according to user's taste i.e. it focuses on personalisation
2. **Small stride** continual learning to improve the base model stored by extension post installation by using some of users' data

Using the small stride continual learning on base model, we will improve the base accuracy of the model itself. However, this would require users' to share their data willingly for the benefit of the community. Users will have the option to not share their data in which case, their data will only be used for large stride continual learning to improve their recommendation model.

3.2.3 Possible removal of back-end component

Currently, the back-end application fulfills two purposes:

1. Train **XGBoost** model using user's features
2. Apply **continual retraining** over the model to increase its accuracy

Both of these purposes could be solved if the **ml-xgboost** library had the support for them. As a future work, we plan to further improve the library by incorporating all necessary support for both of these uses. This would enable us to **get rid of the whole back-end application component** all together and perform all operations regarding training, continual retraining, loading and usage inside the extension itself. This would make the extension much more reliable, secure and fast.

Bibliography

Chen, T. and Guestrin, C. (2016). XGBoost. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM.

Ke, Z., Liu, B., Ma, N., Xu, H., and Shu, L. (2021). Achieving forgetting prevention and knowledge transfer in continual learning. *CoRR*, abs/2112.02706.

Sharma, V. and Mondal, M. (2022). Understanding and improving usability of data dashboards for simplified privacy control of voice assistant data. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3379–3395, Boston, MA. USENIX Association.

Velden, M. and Mörtberg, C. (2014). *Participatory Design and Design for Values*, pages 1–22.