Team 10: *Amnesia* Demo - Recommender System that Can Efficiently "Forget" User Data

Hengyu Tang, Shirley Xu, Nhung Le, Enyi Lian
New York University
{ht1162,xx852,nhl256,el2986}@nyu.edu

ABSTRACT

This project aims to develop a web application to showcase the real-time changes from *Amnesia*, a recommender system that can efficiently and reliably "forget" sensitive user data. The web app serves to demonstrate the corresponding decremental changes of history matrix, total item interactions, cooccurrence matrix, and similarity matrix for an item-based collaborative filtering model after adding or removing the data of a particular user. Specifically, we provide two solutions to interact with Rust and Differential Dataflow so that the model is able to talk to the web application to display the real-time changes.

1 INTRODUCTION

Privacy issues arise as advanced machine learning models are developed using massive personal data. Recent law such as the GDPR (General Data Protection Regulation) in Europe requires the deletion of personal data upon request. Therefore, modern applications that contain ML models aim to remove user data in an efficient and reliable way. It is not sufficient to merely delete the data from primary data stores such as relational databases since the deleted data could be retraced from machine learning models. Retraining a model from scratch without a particular user's data achieves the goal of completely and reliably "forgetting", but it is extremely inefficient and costly from an operational perspective since the model requires access to the entire training data. Thus, a more ideal solution is to decrementally update trained ML models to "forget" the data of a specific user. Our project aims to develop a web application that enables users to interactively "add" or "remove" user data and displays the real-time decremental updates of a movie recommender system implemented in Rust. The web app aims to allow user interactions can demonstrate the changes step by step, thus helping the audience to better comprehend the concept of "decremental updating".

Given our lack of experience in web-app development, we divided this project into four stages to gain incremental improvements and avoid unnecessary road-blocks that may slow down the progress. In the first stage, we aimed to print the model's result in the form of a static JSON file to a web-app. We experimented with different Python-based web frameworks (**Flask**, **Django**), the web development languages **HTML**, **CSS**, and **JavaScript**. Comparing the static web-apps using Python and JavaScript, and taking into consideration complicated next steps of updating real-time results to the web-app, we decided to focus on the Python-based web-app. The second stage is to integrate real-time updates by connecting web applications with model using the distributed stream processing system **Apache Kafka**, the **Cargo** server. The third stage is to integrate user interactions to allow adding and removing users. Given the challenges facing during the second and third stages as mentioned below, we dedicated the fourth stage to optimization methods to reduce updates loading time to the website. In this stage, we implemented **WebSocket** API with JavaScript.

One of the biggest challenges we faced is the deprecation and various limitations when integrating kafka-python| package. For example, although this package is considered to be one of the most popular python interface with Apache Kafka clusters, during our experiments, it presents various limits including

(1) Outdated maintenance incompatible with python 3.7,
(2) Difficulty connecting with other data processing pipelines,
(3) Major defects within the package.

Another challenge is the inconsistency in the results provided by the model, leading to difficulty for the pre-processing stage, and extra time cost to add a temporary buffer for updates in our experiment.

In the end, we were able to create two web applications that meet our initial goals in different ways. The first web application is built using Flask framework, styled with HTML and CSS, and talks to the model via Apache Kafka. Given the drawback of python applications in web-app development, the extensive communication between the web-app and Kafka postponed the result update to the web-app. The second web application styled with HTML and CSS utilizes JavaScript and the WebSocket API that makes it possible to open an interactive communication session between a user's browser and a server so that the website could display real-time updates. As a result, this version allows us to resolve the time-waiting concern and updates the differential data-flow model results in real-time.

2 PROBLEM STATEMENT & APPROACH

2.1 Problem Statement

The *Amnesia* project has been developed to remove user data from trained ML models using decremental updating procedures. The decremental learning process enables models to "forget" user data in an efficient and reliable way. Specifically, we aim to create a web application that demonstrates the decremental updating process for an item-based collaborative filter model. The process can be illustrated by displaying the corresponding changes of history matrix, cooccurrence matrix, total item interactions, and item similarity matrix after adding and removing the data of a particular user. The web application will be initialized with a certain number of users and items and their interactions; the end users of the web application are able to add or remove a specific user each time and submit the request. Once the Differential Dataflow model receieves the request, it will generate updated outputs and transmit them back to the web application so the real-time changes could be displayed.

2.2 Approach

The developing process divides into 4 stages:

- Matrix print by load user-movie interaction initialization and updates from local JSON file. Develop amnesia model output parsing pipelines.
- Integrate real-time updates by connecting web applications with model using kafka server to enable live input feedings.
- Integrate user interactions to allow add and remove users. Cater to edge cases to offer thorough user-friendly experience.
- Optimizations to reduce updates loading time to the website. Resort to connect website to the Cargo server using WebSocket API to bypass python environment.

3 IMPLEMENTATION

(1) Software libraries
  - Flask: Python-based web framework
  - kafka-python: Python client for Apache Kafka
  - WebSocket API for JavaScript

(2) Flask with Python
  - **Stage 1: Print model's results to the web app**. We started with a static JSON sample outputs of the model, and aimed to develop a demo web app that prints four matrices 1) historical data matrix, 2) item interaction, 3) co-occurrence matrix, and 4) item similarity matrix. At each time step, we created one HTML file to load and print the four matrices, as well as a single button for users to print matrices at two time-stamp to show changes. We improved upon the baseline by adding in step-by-step updates of the system using animation with JQuery to better showcase the effect of forgetting user information from the recommendation system. Additionally, we developed parsing process of Rust model outputs, written in Python. Special attention is paid to optimize the process, e.g. we read updates from JSON files only once to reduce memory reading costs.
  - **Stage 2: Update real-time results.**In this stage, we proceeded to connect the web app with the model to present real-time changes of user information. Given the needs to communicate efficiently with rust model and with users, we sought solution in Apache Kafka, which aims to provide a high-throughput, low-latency handling of real-time data feeds. Kafka is picked as it fits our goal and its efficiency, which is implemented to optimize communications between systems to reduce network round-trips.
    We used kafka-python package, the most popular python interface with Apache Kafka clusters to pursue the communication and faced multiple challenges as shown in Part 5.
  - **Stage 3: Add user interface.** We allow modifications on user information and present changes timely. The flask library allows us to extract user entered information into python environmration. We then fed the generated query through kafka-pyhon API to communicate with the model in differential dataflow.
    We used session to store and maintain continuous update of matrices, instead of resorting to external database. Since the amount of data we stored is minimal, we aimed to avoid extra costs of linking python applications with additional platforms.
  - **Stage 4: Optimizations.** The initial implementation dumped model outputs to a designated local file, while the web continues to read in latest updates from that file every time an update command is sent. We hard-coded the running of bash scripts within Python to evoke kafka servers and launch Differential Dataflow. The drawback is obvious as data reading and writing from memory is time consuming, dragging the running performance.
    The second version we redirect output from Kafka Consumer to be captured by subprocess. Consumer object under kafka-python is excluded due to its internal message extraction issue, discussed in detail in Section 5. We adopt multi-threading to ensure consumer running in background to receive continuous updates without getting interrupted every time we reads updates from consumers. The fed-in updates under the same timestamp are pushed to a queue,

and queue contents are cleared once the updates are parsed to reduce the memory usages.
However, due to constant communication between cargo and Kafka clusters, delays cannot be further shrink-ed. Therefore we jump out of the python environment and Kafka to connect differential dataflow directly with the application.

(3) WebSocket with JavaScript
- **Stage1: Initialize the Web App.** Similar to the website built by Flask framework, the web app built with WebSocket in JavaScript also receives the real-time model outputs, which are changes in total item interactions, co-occurrence matrix, and similarity matrix. We updated the HTML display and added the three matrices from the change log of Differential Dataflow, the user-item history matrix from the request submitted in a string of dictionary, and the change log of the original model for review. We implemented a button to "initialize" the website and users can submit the request in JSON format that can be recognized and read by Differential Dataflow. For example, users can submit `{"change":"Remove", "interactions":[[1,1], [1,2]]}` to delete the records of a specific user.
- **Stage2: Add user interface to remove or add.** In this stage, we implemented a more user-friendly user interface. We used HTML Form so that users can input a specific user ID to remove the relevant data. Similarly, end-users can specify a new ID for the user they want to add along with the specific items the new user prefers. WebSocket API allows the browser to send the request to the Cargo server via the `send` function and the browser is able to receive the JSON outputs from the model via `onmessage`, an event listener to be called when a message is received from the server.

(4) JavaScript Angular
- **Stage 1: Load model's results to the web app.** For this version in this stage, we used JavaScript to load model's results to the web app. We started with a static sample output of the model, and aimed to develop a demo web app that prints four matrices 1) historical data matrix, 2) item interaction, 3) co-occurrence matrix, and 4) item similarity matrix. There is only one HTML file that loads the JSON sample output file and creates one dictionary for each time step. For each time-step dictionary, there are four 'keys' to iterate, each key include information of one matrix 1) historical data matrix, 2) item interaction, 3) co-occurrence matrix, and 4) item similarity matrix. There is one 'button' that allows user to switch from time step 0 to time step 1 and vice versa. Given our lack of experience with JavaScript and the time-constraint, we decided to focus on the Python's web-app version and implemented animation functions by using JQuery instead.

Table 1: Minimal Pause for Successful Update Read

Output Capture Approach	Time(s)
Local file writing	5.6s
Threading + sub-process	1.2s

4 EVALUATION

4.1 Experimental Setup

Both websites are implemented under `macOS Mojave` 10.14. Kafka version is implemented under `Python` 3.6. Key package used, together with `subprocess, queue` and `threading`:

- `flask`, version = 0.12.2
- `kafak-python`, version = 1.4.7
- `json`, version = 2.0.9

The WebSocket version is implemented under:

- `differential-dataflow`, version = 0.10.0
- `amnesia_sigmod_demo`, version = 0.1.0

4.2 Datasets

The data we used for developing the model is based on a fixed initialization value and live user input. The fixed initialization history matrix a toy dataset consisting 4 users and 5 movies. We design the live user input data to simulate potential interaction scenarios for practical demonstration purpose, as well as catering to edge case:

- Add new user using movie ID
- Delete user history, both initialized and added users.
- Continuous deletion of user until one movie has no interactions.
- User input mistakes, e.g. entering wrong movie names.

For better user experience, we maintain both usernames and movie name dictionaries to allow entering of new user history more naturally.

4.3 Results

The final web demo are implemented in 2 versions: one based on Flask and Kafka, another using JavaScript and web-socket. The final Flask application requires manual start of Kafka and Cargo server to reduce program loading time. It fetches model outputs using sub-process and multithreading directly from KafkaConsumer instead of going through python API. The reduction of program pausing time for successful update read is shown in Table 1: We further made efforts to improve user experience by attending to details. The users are free

to enter both real movie (user) name listed and movies(user) index since requiring movie index starting with 0 will create extra confusion for new users. We also alerted the scenario when user entered the wrong movie information, but web presentation will be unchanged.

The web application implemented in JavaScript, on the other hand, is able to display real-time changes. The updating time is too minimal to properly measure. WebSocket API allows efficient communication between a user's browser and the Cargo server, so we focused more on creating a more user-friendly interface. The end-users of the web application could submit the complete requests in strings or specify user and movie IDs before clicking the "Add" or "Remove" button. Moreover, we incorporated more designed elements in the interface, such as button color change on hover and default text for HTML Form placeholder. However, given the limited time we had, we were unable to handle invalid user inputs. In addition, we can improve the interface by enabling users to input movie titles rather than movie IDs.

5 DISCUSSION

(1) **Observations**
Python here proves its strong ability in extending with other developing systems, emphasizing its flexibility. Besides providing a easier transition for newcomers to quickly generate a working website, python also excels with its powerful flask package. Flask ensures the flexibility by allowing data processing under python environment, which indirectly connects web-app development to vast package resources. For example, we are able to connect with Kafka cluster, which is difficult to achieve so using JavaScript. However, the process also exhibits python's drawbacks, which is a common issue faced by python based approaches. The constant communications between different developing environments, as well as python's slow nature will place a big burden in website running performance. On the other hand, combination of JavaScript and web-socket provides fast processing and communications with severs. Also Web-socket with JavaScript has an easier setup steps and allows development of web-app under an integral environment.

(2) **Challenges**
One of the biggest challenge we faced is the deprecation and various limitations when integrating kafka-python package. Although this package is considered to be one of the most popular python interfaces with Apache Kafka clusters, during our experiments, it has various limits: 1). Outdated maintenance incompatible with python 3.7, 2). Difficulty connecting with other data processing pipelines (i.e., unable to connect the API with cargo under python environment, leading to failure in communication with the distributional dataflow, 3). Major defects within the package (e.g., given a Consumer instance initiated, it cannot be properly assigned to Kafka topic specified hence cannot print out message directly.)

(3) **Future Topics**
One future topic includes maintain and updates of the current kafka-python package. We believe it is a highly valuable package to connect python program using kafka clusters. Potential goal includes fix KafkaConsumer topic subscription failure and migrate to be compatible with Python 3.7
Potential future topics includes demonstrations of other amnesia models especially k-means. A visual step by step change of clusters distribution after removing certain user information will showcase the amnesia effect clearly and effectively. We believe this topic is highly valuable, as clustering is also a common strategy executed by the companies for commercial purposes. New items usually will go through a profiling stage, where its behavior is compared with existed items to alleviate cold-start problems, to assist the company make more targeted yet more inclusive recommendations.

6 DETAILED CONTRIBUTIONS

6.1 Hengyu Tang

Implemented the WebApp using Flask framework, styled with HTML and CSS, and integrated with Apache Kafka.

6.2 Shirley Xu

Implemented the WebApp using WebSocket API, styled with HTML and CSS. Apache Kafka investigation.

6.3 Enyi Lian

Implemented the WebApp using WebSocket API, styled with HTML and CSS. Apache Kafka investigation.

6.4 Nhung Le

Implemented JavaScript Angular framework, styled with HTML and CSS. Apache Kafka investigation.

7 REFERENCE

1. Schelter, Sebastian. "Amnesia" – Towards Machine Learning Models That Can Forget User Data Very Fast.
2. Jun, Rao. "Open-Sourcing Kafka, LinkedIn's Distributed Message Queue." LinkedIn Official Blog, LinkedIn, 11 Jan. 2011, https://blog.linkedin.com/2011/01/11/open-source-linkedin-kafka.