Anglia Ruskin University

Course: Applied Data Analysis and Research Methodology

Module Code: MOD007893

Student ID: 2063656

Project: Harassment Tool Deployment Simulation

Table of contents

Abstract

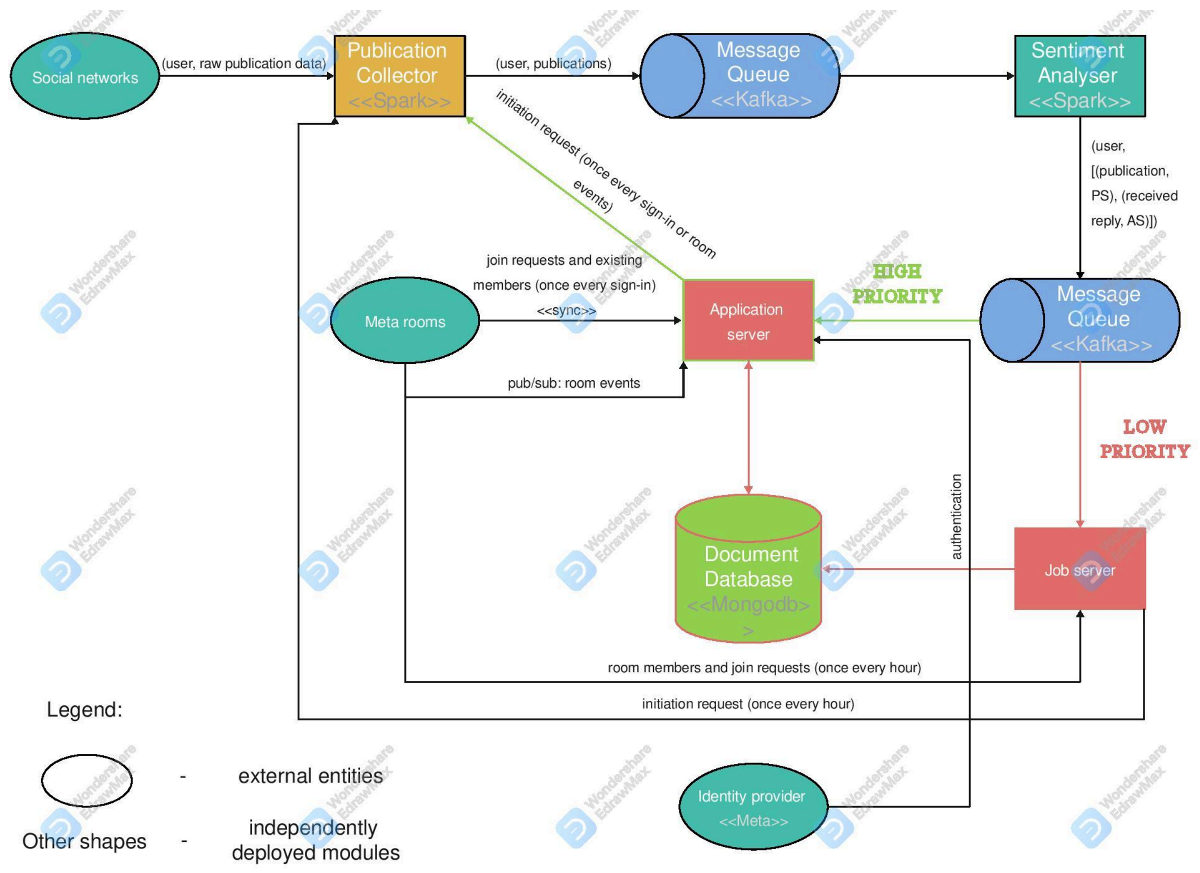
In this project, since I do not have access to the premium tools that were at the foundation of the initial concept detailed in Assignment 010, I decided to either replace them with simpler, but nonetheless functional alternatives, or to simulate them by randomly sampling data from datasets that I was either provided or found online.

More specifically, having access to an API that retrieves Metaverse users' publications on traditional social media was a major underlying assumption, and one that could likely only be realised by having professional tiers with Meta. Therefore, instead of retrieving the publications of real internet users, my solution was to retrieve these comments from virtual users, simulated using a variety of Python techniques including classes, threads, random sampling, and others. In addition to obvious simulated behaviours of having the tendency to post comments on, and receive replies from, various social networks, as well join, request access to, and leave Metaverse rooms every now and again, the virtual users were given names and, naturally, sentiment scores to help the user of MAY identify potential cyberabusers.

In addition to simulating users, I also built, trained, and compared several sentiment analysis engines and, through careful selection of what data to include and what not to, was able to obtain better results than the widely used spacy sentiment analyser, without having any information about the data the engines were tested on.

Finally, like proposed in the 010 Assignment, I included a proof-of-concept for detailed analysis of each of the users' behaviour.

Data pipeline


Social networks
(user, raw publication data)
Publication Collector <<Spark>>
(user, publications)
Message Queue <<Kafka>>
Sentiment Analyser <<Spark>>
(user, [(publication, PS), (received reply, AS)])
initiation request (once every sign-in or room events)
join requests and existing members (once every sign-in) <<sync>>
Meta rooms
Application server
HIGH PRIORITY
Message Queue <<Kafka>>
pub/sub: room events
LOW PRIORITY
authentication
Document Database <<Mongodb>>
Job server
room members and join requests (once every hour)
initiation request (once every hour)
Legend:
- external entities
Other shapes - independently deployed modules
Identity provider <<Meta>>

Idealistically, the high-level architecture of the application is conceptualised as above. This is what the application's bird's eye view will be like if it ever gets a public release. However, as mentioned above, there were a number of constraints on both the software and the hardware that was available at my disposal; therefore, I need to replace many of the components with less costly and computationally demanding alternatives. I believe I have succeeded in replicating the entire data pipeline (except the optional identity provider module) via free libraries available in Python alone.

The social network component was replaced with a thread which randomly samples comments and replies from pre-loaded datasets. In fact, these comments and replies were actually made by real internet users, and nothing more than the content of these comments and replies, as well as their sources, were sampled. Therefore, the effect of the replacement is more or less identical to having an API that collects publications in real-time — just like originally intended — but with a delay.

Similarly, the metaverse rooms were simulated using custom-built Python classes, instances of which were randomly filled with the simulated users (who were also randomly assigned a room status — either "room member", "room join request", or "room non-member) and represented the metaverse rooms.

The publication collector was replaced with a second thread which gathered the data from the social network replacement component using a custom-written function.

The Kafka message queues were replaced with free message queues importable directly through Python.

I have devised three separate sentiment analyser modules, two of which I constructed on my own via training classification models on data from the provided datasets (refer to the “training methods” section for more details) ─ as well as an additional, much larger dataset that I found online ─ through vectorisation. Moreover, both of these performed better than the pre-templated engine from Spacy that I also tried on brand-new data. Unfortunately, due to the hardware constraints on Google Colab’s servers, I wasn’t able to use my more effective models ─ which were, in addition to being more effective, faster ─ without Google Colab occasionally crashing. Hence, I used Spacy’s pre-templated engine as a replacement for the module.

The job server was simulated via a collection of Python lists, which collected data from the sentiment analyser and passed the data forward to the database module.

The database module, in turn, was emulated using some more Python lists, which were being constantly appended and modified as the threads kept running and as data kept flowing through the pipeline.

Finally, the application server output was emulated via print() commands and an array of Matplotlib plots containing several visualisation elements: two pie charts demonstrating the distributions of ARAS and APPS scores, respectively, of all users in the system as well as four bar charts displaying the distribution of ARAS and APPS scores of a randomly selected user by two different criteria: data and source. Thus, together, the application would output 6 distinct visualisation elements roughly every 10 seconds (the refresh rate chosen as per the Non-Functional Requirements section in the 010 assignment). Note that all the visualisation elements are displayed just like originally described in the 010 assignment with only minor functional changes.

Training methods

To train my two sentiment analyser modules, I used a variety of data fusion techniques in combination with the established standards for sentiment analysis training to produce two models which analysed the sentiment of input text even better than the popular sentiment analyser Spacy, which was trained on similar data.

One data fusion technique I made use of was the concatenation of different datasets ─ two datasets containing Twitter comments, one dataset containing Facebook comments, and one dataset containing Amazon comments. This enabled greater generality of the eventual sentiment analysis models, and hence higher reliability in evaluating new data.

Another data fusion technique was the comparative search over 3 different models ─ one trained using a neural network, one trained using Naive Bayes, and one template analyser from Spacy; the Naive Bayes model ended up the strongest performer when all 3 models were tested on new data. This comparative search allowed me to select the classification algorithm on an objective basis rather than having to resort to manually picking what I subjectively thought was the best one.

As to the training procedure, it was as follows:

1) First, I split my concatenated dataset into train data and test data so that my models would be tested on data they weren't trained on, and so that their performance on the test data would hence be an unbiased metric of their effectiveness.
2) I then converted each of the comments in the dataset into sequences of tokens, with each token consisting of either a word or special symbols such as brackets and colons. The inclusion of special symbols proved to vital to the ultimate performance of the models, and increased the accuracy of the best performer, the Naive Bayes model, from 19% to over 70%. This is likely due to the presence of emoticons such as ":)", which are great indicators of the sentiment.
3) Consequently, I removed all the stopwords (like "the" and "a") from the tokens, as they provide almost no information about the sentiment of the input text, take up a lot of computing power, and may end up confusing the models. I also converted all the comments into lowercase, as the case of the letters likewise provides almost no information about the overall text sentiment.
4) I recorded all instances of a token across the training data in a list representing the vocabulary of the models that I would later train.
5) Finally, I transformed each of the token sequences into vectors that represented whether each of the token instances in the vocabulary was present in said sequence.
6) To ultimately train the model, I just fit my classification algorithms onto the array of vectors from step 5).

Evaluation

This project can roughly be divided into three parts: simulation of data pipeline components I didn't have access to (such as live user data, Metaverse rooms, etc), sentiment analysis engine building, and MAY application building.

I believe that incorporating the simulation part was a creative solution to the problem of lack of access to MAY's functional requirements. It's a step that I could have avoided completely and have simply skipped right ahead to the latter two parts; however, I devised and implemented it for maximal closeness to the original proposal, and it ended up functioning just as intended.

As to the sentiment analysis part, I combined a variety of data fusion techniques with industry-recommended practices to build an effective sentiment analysis engine. However, I was not able to use this engine, which is an area I could improve on.

Finally, the MAY application part also functions just as intended, outputting most of the visualisation elements described in the 010 assignment.

An important advantage of simulating the original concept rather than reproducing it — which wouldn't be possible for reasons outlined in the Introduction section — is the fact that security, privacy, and confidentiality don't have to be accounted for. Moreover, the solutions to these issues largely depend on the nature of the APIs listed in the Functional Requirements section of the 010 assignment, which I don't have access to; therefore, I believe that all these issues are outside the scope of this project.

Nevertheless, there is still room for improvement in the project. In particular, instead of Google Colab, I could have used an alternate, more powerful Python application, which would have allowed me to use my custom-built sentiment analysers that would produce more accurate sentiment scores. Moreover, while I tried to stick to the original proposal detailed in the 010 assignment as closely as I could, there are some places in which, in theory, I could have stuck to it even closer. For example, the original proposal suggested drawing pie charts demonstrating the breakdown of sentiment scores by publications made by, and in reply to, each of the users in the system. However, instead of that, I decided to draw a pie chart showing the breakdown of aggregate sentiment scores by user. The decision was justified by my desire to diversify the scope of my data visualisation elements, which would otherwise all be user-specific; however, as mentioned, the drawback of this decision does deviate from the original proposal.

Conclusion

Despite the lack of resources and hardware constraints, I believe I have succeeded in producing a functional prototype of the original proposal, demonstrating a proof-of-concept for the idea. I had to cut back on the number of operations per second all throughout the project; however, I believe it still constitutes a functional application quite close to the original concept.