A Toolbox for Reducing Mental Health Issues Caused by Common Social Media Features

Contents

Introduction

This toolbox serves as a fundamental resource for any social media based software developer aiming to effectively consider the general mental health of their platform's user base.

With ever growing developments in social media as an industry (for example, the idealised concept of virtual worlds based on social connection, namely *Meta*[1]), the focus on encouraging digital social connections between users can sometimes be so prevalent that the negative, social repercussions of this encouragement can be ignored, namely mental health issues. As a software developer, these issues are rightfully difficult to accommodate for and mitigate. The issues can sometimes be intractable, such as sociological and psychological issues, where no algorithm or computation exists to solve them. Hence, this toolbox serves as a resource for software developers wishing to combat the negative effects their social media platform has on their user base's mental health.

To achieve this, the toolbox is structured around providing the reader with information about a number of thoroughly researched, adaptable tools for reducing or eliminating the damage done to a user's mental health by common social media features, specifically:

- **Reducing a User's Screen Time** - To combat the addictive design of social media platforms that evokes a sedentary, unhealthy lifestyle in their user base.
- **Improved Moderation** - To improve the existing method of moderating user-based content, providing a tool to reduce toxicity, bullying and the presence of hateful communities in social media platforms.

For each tool, the general steps of technical implementation are explained and detailed with the goal being that these instructions can be followed by the reader to adapt the tool for any social media platform or context. Additionally, code-based demonstrations, commented code and visualisations accompany the step-by-step implementation instructions where necessary, to further explain a tool's functionality and help explain its concept.

Each tool has also been fully implemented on at least one well suited platform to serve as a full technical reference. Each of these implementations can be found in the following Github repository https://github.com/Squadrant/Humane-social-media-guide. The directory structure can be seen in the image below. The reader is encouraged to use, adapt, and integrate the implementations into their own platforms.

[1] https://about.facebook.com/meta/

```
Mental-Health-Toolkit-for-Social-Media
├── Notification Postbox Tool
|   └──PostboxNotification                    (the Android Studio project directory)
|       └── app/src
|           ├── main
|           |   ├── java/com/squadrant        (java source files)
|           |   └── res                       (resources directory)
|           ├── test/java/com/squadrant       (unit tests)
|           └── androidTest/java/com/squadrant  (instrumented tests)
├── Spam Filter Tool
|   ├── Spam Filter Model Builder.ipynb
|   ├── Notification Predictor.ipynb
|   ├── Example Inputs and Outputs
|   |   ├── Example Notifications.csv
|   |   ├── Spam Filter Model.joblib
|   |   └── Notification Prediciton.csv
|   └── Testing
├── Exit Points Tool
|   └── Exit_Points.ipynb
├── Time Visualisation Tool
|   └── Time_Visualisation_Tool.ipynb
├── Moderation Tool
|   ├── moderation_model.ipynb
|   ├── Visualisation.ipynb
|   └── Image_to_Text.ipynb
└── Guide.pdf
```

As can be seen from the directory structure of the toolbox, most tools have been developed as Python notebooks (.ipynb files). This is because Python is easy to read, has a lot of supported libraries, and is very portable because of services like Google Colab. The simplest way to use these tools is to upload the Python notebook files into a Google Drive, and then open the files in Google Colab and run them according to the guide instructions for the relevant tool. The tools can be run in any Python notebook editor, e.g. Jupyter Notebook, though they were developed using Google Colab exclusively.

The one exception to the Python implementation style is the Notification Postbox Tool. This is a user facing feature and so was developed as an Android application. More details can be found in the Notification Postbox section of this guide.

Reducing a User's Screen Time

A common feature in social media platforms is their addictive design; they are developed in specific ways to keep the user engaged on their platform for as long as possible. From a business perspective this feature is justified, as a platform's revenue model is usually related to how long a user spends browsing a platform and interacting with content: such as advert revenue. However, these design choices can have a negative impact on the mental health of a regular user.

Addictive design choices encourage addictive interactions between a user and a social media platform which in turn encourages a sedentary lifestyle, as a user develops an unhealthy reliance on the addictive cycle provided by the platform. This sedentary lifestyle has links to poor mental health.[2] [3]

The tools provided for this issue aim to minimise the effects of addictive design, whilst still maintaining the benefits for a platform's business case.

Specifically, this toolbox provides the tools to combat these identified addictive design features:

- **Notifications as an Addictive Trigger** - For mobile based social media platforms, notifications can act as a means of luring users back into spending time on a platform when the user was not originally intending to spend time on the platform. For example, if a user receives a notification that a friend has uploaded a picture, this notification is not urgent but could feed the addictive loop by encouraging the user to use the platform and spend time browsing content.
  For this feature, the toolbox proposes a **Notification Post Box** and a **Notification Spam Filter**. These features are explained in detail in the corresponding sections of this guide.
- **Endless Content** - Many platforms deny users 'stopping points' whilst browsing platform content. When content is constantly fed to a user in an endless scroll, they struggle to find a reason to stop using the app. Consider the principle of inertia - users will keep scrolling unless triggered to stop. This design to keep users browsing content feeds into an addictive cycle.
  For this feature, the toolbox proposes the use of **Exit Points.**
- **Hiding a User's Interaction Time** - Platforms fail to be transparent with a user's browsing data, and fail to work with users to help them manage their own time spent browsing different types of content.
  For this feature, the toolbox proposes a means of **Time Visualisation and Control.**

[2] Hoare, E., Milton, K., Foster, C. et al. The associations between sedentary behaviour and mental health among adolescents: a systematic review. Int J Behav Nutr Phys Act 13, 108 (2016). https://doi.org/10.1186/s12966-016-0432-4

[3] Teychenne, M., Costigan, S.A. & Parker, K. The association between sedentary behaviour and risk of anxiety: a systematic review. BMC Public Health 15, 513 (2015). https://doi.org/10.1186/s12889-015-1843-x

Notification Postbox

Goal

Notifications are a very common feature in mobile applications. This is a proof of concept demonstrating an alternate method of handling notifications, with the secondary purpose as a tool for end-users to corral notifications from other apps. This tool is implemented for the Android operating system due to the ease of development on the platform and how widespread it is, though the design ideas are broadly applicable to all mobile applications.

Glossary

The Android operating system has several existing mechanisms for users to manage notifications from your application.

1. **Do Not Disturb** mode allows users to block all notifications from making sounds, though they appear in the system UI as normal unless otherwise specified. The user can also allow certain types of notifications to interrupt them (alarms, reminders, events, calls, and messages).

2. **Notification Permissions** for the app can be configured. At the broadest level this sets the maximum interruption an app is permitted to give the user. This can be done at the app level, or, since Android 8.0 (API level 26), for different types of notification.

For more information on notification specifics for Android consult the documentation page[4].

Use Case

Any app that uses notifications can benefit from considering how those notifications are received by users. By allowing users to tune notification settings and behaviours within the app itself you can offer finer control compared to the android system settings and better standardise across platforms. The android system settings are also difficult to find and so you offer better control to users.

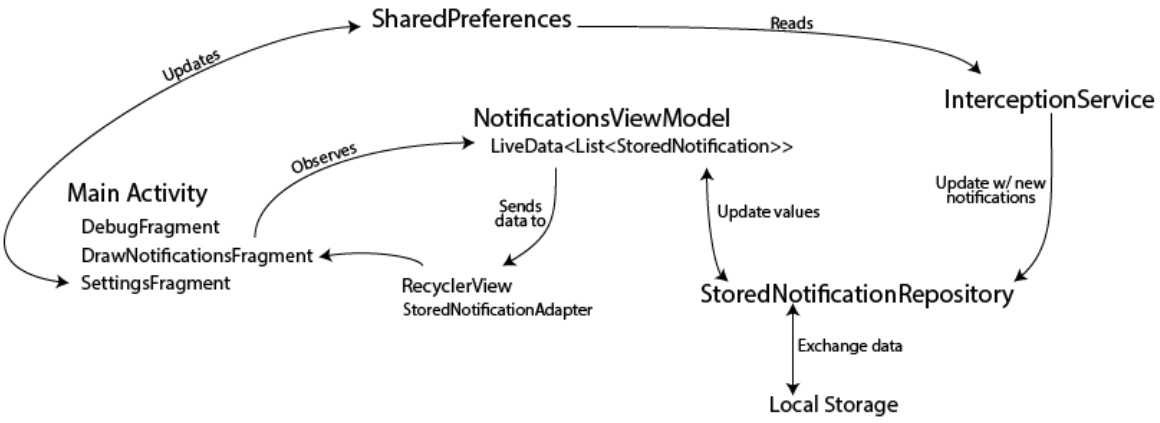
Whilst the exact techniques for this proof of concept may not be applicable to your app the principles of user control are universal and bear consideration.
The actual proof of concept app developed is aimed primarily towards users who are seeking to better control notifications than as a developer example because notification use is so specific to the platform, though some aspects may be helpful for developers.

Technical Overview

The app is implemented using the typical MVVM architecture (Model, View, View Model) which separates the UI, data, and data representation into separate classes. In android this is important for performance reasons and due to the lifecycle of UI elements being unpredictable and short - the operating system frequently recycles and resets them e.g. recreating the UI on phone orientation change. Apps with more complex behaviours may

[4] https://developer.android.com/guide/topics/ui/notifiers/notifications

require a more complex architecture[5] but the components of MVVM are widely used.



The UI is controlled by the MainActivity which hosts Fragments. The three fragments are navigated to by the user using a bottom navigation bar[6]. The debug fragment allows the generation of notifications for testing purposes. The settings fragment uses the AndroidX Preference library[7] to store user settings. The draw notification fragment unsurprisingly renders intercepted notifications.

Actual notifications are modelled using the StoredNotification class that stores a subset of android notification features. When implementing a similar system you will likely require additional fields for more complex actions (e.g. actions or images), but since the system notifications represented are not owned by the app it is not possible for those additional features to be extracted; a problem that you do not have for your own notifications.

Since notification data is stored to be used by the app, it is logical to have a class responsible for maintaining a single source of truth (SSOT)[8], this is the StoredNotificationRepository class. This class uses the local storage allocated to the app as its backend though this could easily be a remote or local database or both. The mechanism by which the repository stores the data is not relevant so long as it fulfils its contract as a SSOT.

So far the UI and data layers have been described, covering the model, view, and repository structure. The glue that links the two is the view model. The view model's lifespan is the same as the app meaning it persists over layout changes that cause UI elements to lose data. It maintains a LiveData object. LiveData objects are observable, that is the UI elements subscribe to the object and when the data changes (e.g. a new notification is added to the repository) a callback specified by each observer is executed. This callback is used to update the UI so that it matches the updated data. Therefore the view model interacts both with the UI (via DisplayNotificationsFragment) and with the repository (needs to ensure its data is concurrent with the SSOT).

The repository interactions are simple. The repository class has a live data that the view model gets a reference to, therefore when the repository processes any data update it

[5] https://developer.android.com/topic/architecture
[6] https://material.io/components/bottom-navigation/android
[7] https://developer.android.com/guide/topics/ui/settings
[8] https://en.wikipedia.org/wiki/Single_source_of_truth

can propagate those changes to the LiveData variables outwards - fulfilling its role as the single source of truth. The UI also uses the view model to propagate changes to the repository, for example when cancelling a notification. The view model simply calls the corresponding repository method. This keeps the modules as decoupled as possible - the UI does not know about the repository, only the view model.

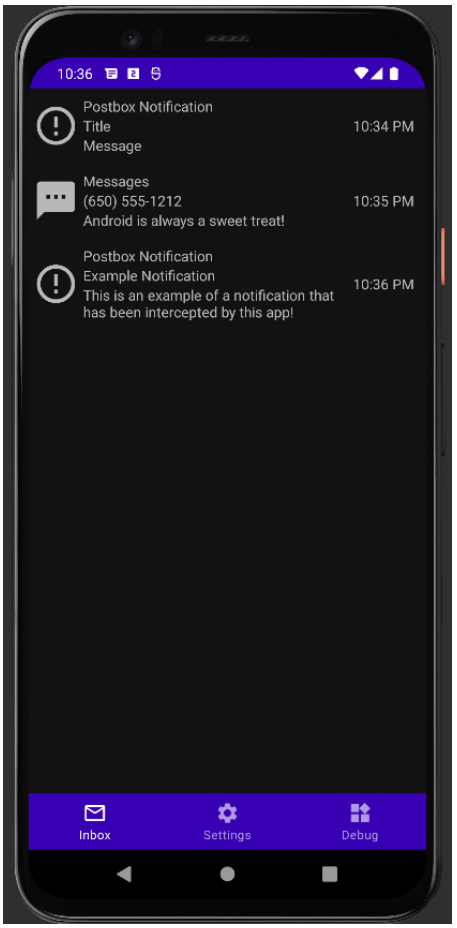
The view models interaction with the UI is a little more complex. The stored notifications are displayed in a recycler view[9], which is a container view for lots of data with each element having a view. The recycler view requires an adapter to convert each StoredNotification into a view for display. So when the LiveData updates, the callback creates a StoredNotificationAdapter. The adapter then, for each stored notification, inflates a layout and uses the notification to fill in the details (app title, notification content, notification time, etc). The callback also sets up swipe behaviour to delete notifications (via the view model) and to undo deletion through a snackbar[10].

The final core element of the app is the NotificationListenerService. This is not necessary for the displaying of notifications, so if you are implementing an alternative notification display system this is not pertinant, but is included for completeness. When a notification is intercepted it checks the preferences the user has specified (in the SettingsFragment) to see if it should first store the notification and then if it should suppress the notification from reaching the system UI. It also listens for notification cancellations as the user may also specify that if a notification is cancelled on the system UI but is also stored by the app then the app should delete its copy.

[9] https://developer.android.com/guide/topics/ui/layout/recyclerview
[10] https://developer.android.com/training/snackbar/showing

Demonstrations

An example of some intercepted notifications if shown to the right.



The app itself can be built and installed as any android application can, provided the device has at least SDK 27 (Android 8). The app was developed using Android Studio and so the source is best modified using Android Studio, though alternatives will work with slight adjustment.

The specific nature of how you need to present your notifications and the level of setup you require will vary heavily, though this example as well as the linked documentation should be beneficial.

For users of the app a bundle or APK can be built for installation or app store hosting, after which the app will prompt the user for requisite permissions (for the listener service). After that the app is fully functional and installed. Android Studio easily allows this from the provided source code.

Further Considerations

This particular task is more user facing than most of the tools here as its goal is not developer facing; while there may be some utility in considering how notifications should be handled the code developed is not likely to port to domain specific needs particularly well. However for users it potentially provides a useful tool for managing notifications and app interactions where existing measures are lacking or difficult to use.

Notification Spam Filter

Goal
This feature is a trained model that predicts whether a notification is important or unimportant to a user, similar to a spam filter used in an emailing system. The goal of this feature is to be a tool that can be used to stop the notification trigger in the habit cycle of overusing social media. This would be done by reducing the number of push-notifications a user receives, by blocking unimportant notifications.

Glossary
- Joblib File - joblib[11] is used by this tool to store the Python Machine Learning models produced by this tool as joblib files, to allow for the models to persist and be used without retraining the models from scratch each time a prediction is needed.
- Notification Channel - different types of notifications are grouped into channels, for example ‘Tags’ or ‘Comments’. Some users find certain types of channels more important than others. For this reason, the notification channel is used as a feature in this tool.

Use Case
This tool is a backend feature. An example use-case for how this tool could be used is as follows.
When a push-notification is sent out by a social media application, it would be intercepted and its details passed into the predictor model. Based on the notification channel and the names in the notification, the model predicts if it is important or not to the user.
- If it is deemed not important, the system can then block the notification from alerting the user.
- If it is deemed important, the system can allow the notification to be sent.

The model is to be trained and used on a per-user basis because notification importance is subjective to each user. The performance of a generalised model would be difficult to test since it could vary so much from user to user.

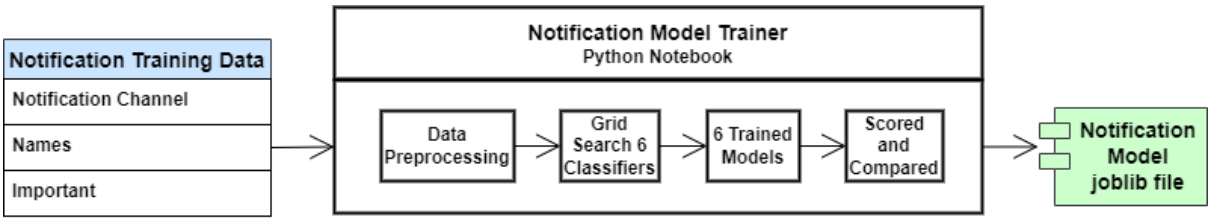
As will be explained later, the model will need to be re-trained regularly to ensure optimal performance.

Technical Overview
The predictor was implemented using Google Colab’s Python notebook. To load the relevant files into the notebooks from Google Drive, a cell is provided to mount a Google Drive directory, giving the notebook access to the files in that account. However, as already mentioned the code can be used in any Python notebook editor, e.g. Jupyter Notebook. The Google Drive mounting cell can simply be deleted in that case.

This predictor tool is split into two parts: training and using.

[11] https://joblib.readthedocs.io/en/latest/

Training the model



Spam Filter Model Builder.ipynb takes existing notification data and outputs the best model to a joblib file (see the image above for a visual breakdown). The notification training data must be a CSV file with a column for the notification channel, a column for the names involved with the notification (e.g. tagged in a post and by who), and a column for whether the user found this important or not. This importance could be measured by whether they opened a notification in the app. The notebook trains six different models and scores them using 'score = 2*Sensitivity + 3*Specificity'; the model with the highest score is outputted as a joblib file.

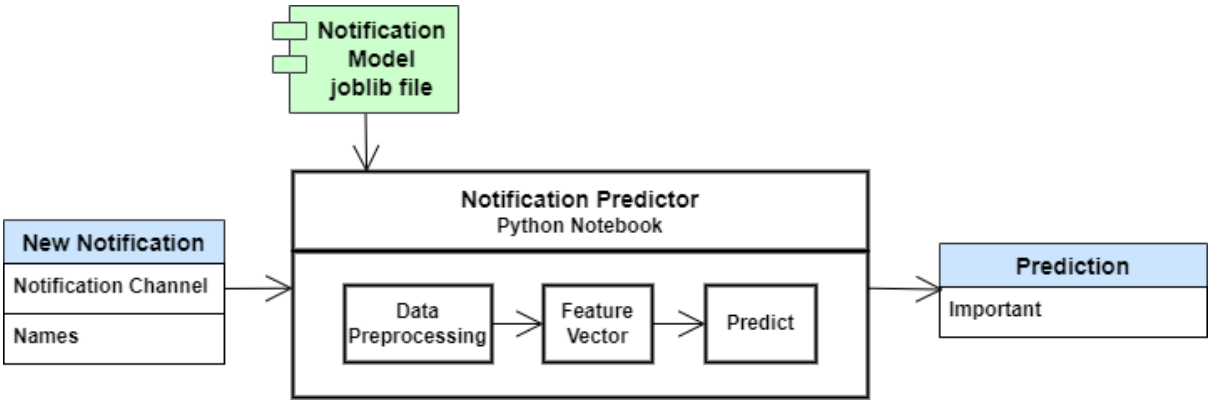
The two metrics, sensitivity and specificity, are unevenly weighted in the score calculation. Our customer felt that it was more important to block unimportant notifications (be specific) than to let important notifications pass through (be sensitive). This is because notifications are inherently not *vitally* important and so, improving mental health and perhaps missing a few interesting post notifications is preferred to sending the user every interesting post as well as some uninteresting ones that will needlessly trigger the user to scroll on social media. The goal of this tool's use is to only send notifications when it is important (i.e. only trigger the user when it is something they will likely find interesting).

Since each user would have a different model trained on their own data, it is impractical to manually check which model performs best for their data. For this reason, the notebook trains six classifiers (each with a grid search performed to find the best performing model parameters), and compares them using the score function described above. The six classifiers used in the notebook are:

1. sklearn.ensemble.HistGradientBoostingClassifier()
2. sklearn.neighbors.KNeighborsClassifier()
3. sklearn.tree.DecisionTreeClassifier()
4. sklearn.svm.SVC()
5. sklearn.naive_bayes.BernoulliNB()
6. sklearn.neural_network.MLPClassifier()

The best scoring model is outputted to a joblib file.

Using the model



Notification Predictor.ipynb loads in the joblib file's model and provides an interface to take a new notification and then outputs whether it is likely to be important or not (see the diagram above for a visual breakdown of how the model is used).

Guide

Training the model

What you need:

- *Spam Filter Model Builder.ipynb*
- CSV file with at least the three following columns, named exactly as follows:
    - 'Notification Channel' - taken straight from the notification.
    - 'Names' - taken from the notification separated by periods e.g. 'Forename Surname. Forename2 Surname2'. These names are who was involved in the notification e.g. *who* tagged you in a post.
    - 'Important' - 'T' if the notification is important, 'F' if the notification is not important.
    - An example CSV file, *Example Notifications.csv,* is included in the Example Inputs and Outputs directory. For ease, an additional example for the expected layout is shown in the image below.

1	Names	Notification Channel	Important
2	Afore Asur. Bfore Bsur	Comments	F
3	Bfore Bsur	Updates from friends	F
4		Reminders	T
5	Cfore Csur. Dfore Dsur. other people	More activity about you	T
6	Efore Esur. Cfore Csur	More activity about you	F
7	Cfore Csur	Tags	T
8	Ffore Fsur	Events	T

How to train the model:

1. If using Google Colab, run the first code cell. If not, ignore/delete it.
2. To load the training notifications CSV file, change the path in the second code cell to your training CSV file path.
3. Run the rest of the cells and the notebook will output a joblib file containing the best trained model.

Using the model

What you need:

- *Notification Predictor.ipynb*
- A joblib file containing the trained model outputted by *Spam Filter Model Builder.ipynb.* An example joblib file, *Spam Filter Model.joblib,* is included in the Example Inputs and Outputs directory.
- Exactly one notification to predict upon. The following fields can be loaded in as a CSV file or inputted manually:
    - ‘Notification Channel’ - can be taken straight from the notification.
    - ‘Names’ - taken from the notification and separated by periods: ‘Forename Surname. Forename2 Surname2’. ‘nan’ can be used to mean no names.

How to use:

1. If using Google Colab, run the first code cell. If not, delete it.
2. To load the trained model, change the path in the second code cell to your model’s joblib file path.
3. Manually change the notification data (period separated names and a channel name).
   Or
   Uncomment the relevant cell and change the path to your notification’s CSV file path.
4. Run the rest of the cells. The notebook will output a CSV file containing the notification prediction.

Maintaining the Model

If the new notification uses a channel that the model has not been trained on, or names which were not trained with, then those features are not added to the feature vector for prediction in *Notification Predictor.ipynb*. This is because the new feature vector for the prediction must match the format of the training feature vectors. The format of the training feature vectors is with the names and channel names being binary features of whether or not they are present. Visually, the training data is reformatted from

Row	Notification Channel	Names
1	Comments	Bob Bobbington. Charles Charlington
2	Tags	Emma Emmington. Bob Bobbington
3	Tags	Charles Charlington

to

Row	Bob Bobbington	Charles Charlington	Emma Emmington	Comments	Tags
1	1	1	0	1	0
2	1	0	1	0	1
3	0	1	0	0	1

with each category being a binary column. This was necessary due to the original columns being categoric.

As a result of this formatting, the model cannot make predictions with unseen categories (e.g. ‘Fred Fredington’ in this case), since ‘Fred Fredington’ would not be a column in the training feature vectors. The predictor simply ignores the presence of the new category(s), and makes a prediction based on the presence of the features from the training stage.

This limitation of the predictor is why the model should be updated regularly. For example, each month, store the data of whether a user opened a notification in-app or not (not opening it being synonymous with it not being important to the user). At the end of the month, re-train the model on the new data, and make predictions for incoming notifications based on the new model. This will help to ensure the model has an acceptable performance.

Further Considerations

The notification filtering feature developed in this project is a backend tool for predicting notification importance. The natural next steps are to link this to a mobile application that can block the android notifications predicted as unimportant. Some resources for this next step are provided in this section.

Notification Postbox

The notification postbox in this guide shows functionality for capturing and blocking android notifications.

Deploying Machine Learning Models In Android Apps Using Python
https://analyticsindiamag.com/deploying-machine-learning-models-in-android-apps-using-python/

Deploy a Python Machine Learning Model on your iPhone
https://towardsai.net/p/machine-learning/deploy-a-python-machine-learning-model-on-your-iphone

Exit Points

Goal

It is often hard for some users of social media to stop using a social media platform well after they had intended to use it. This can lead to an unhealthy relationship with social media, with the worse case being social media addiction. This unhealthy relationship is facilitated by the design of the social media platform. A constant stream of new content is provided to the user, and the user is constantly searching through this content to find something they enjoy. This creates a situation where a user continues to scroll through content for a much longer time period then they intended to, while gaining no additional satisfaction.

By breaking this stream of content up by using exit points we can disrupt the stream of content and give the user a clear opportunity to assess their use of the platform and if they have engaged with it as much as they wanted. If they have, they can break the content-seeking behaviour and then spend their time on other things. This helps encourage a healthier relationship with social media, one where the platform respects the users' time and does not try to place them in a skinner box to maximise time on the platform.

Use Case

This tool requires being able to keep track of the number of posts a user interacts with or goes past. It is best suited for social media platforms that otherwise would provide a constant feed of recommended posts that the user can stop scrolling and look at or ignore and continue scrolling to more content. It is intended to break up the feedback loop often present in social media streams to encourage users to not endlessly scroll.

It is most appropriate where a virtual skinner box is likely to exist, such as on platforms with short pieces of content that the user scrolls one by one through. This can easily become a skinner box situation for a user, as the user is unsure when the next video they really enjoy will come up in the feed so feel the need to keep scrolling for far longer than they originally planned to engage with the social media platform.

Technical Overview

What an exit point looks like will depend on the platform and design decisions made to best fit within the social media platform this feature is developed for.

Some example exit points are having a pop up asking the user if they want to keep scrolling, having a special buffer content which appears in the a users content feed, a user manually clicking a button to load more content when they have been recommended a set amount of content. There are many other possible ways to implement an exit point but the key part of an exit point is creating a situation where when an exit point occurs in a user's experience they can easily stop using the social media platform.

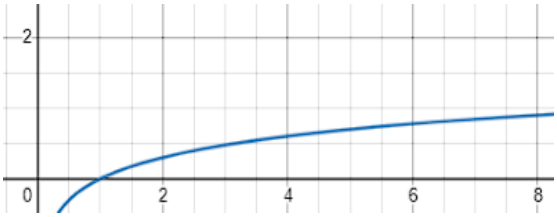
An important consideration with exit points is the frequency that they occur within a user's experience. Having exit points too often can drive users away as if they want to engage with the content on the platform then they have to get through an excessive amount of exit points to get to what they want to get to. Having too little exit points would mean that the users would have large amounts of time wasted, which exit points aim to solve.

To avoid these situations the right frequency of exit points is required. The simplest way is to decide an appropriate constant frequency so that after every set number of posts a user sees, there will be an exit point. The main problem with this implementation would be that a user will likely be more frustrated by exit points at the beginning of their usage of a platform

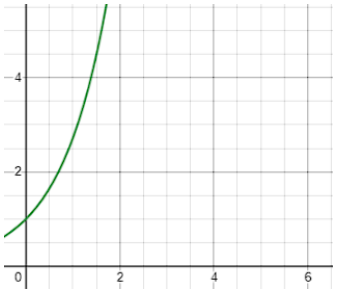
than later on, so tuning the frequency so that the user is not annoyed by exit points at the start of their usage would likely make the exit points too infrequent by the end of their usage. By making the frequency of exit points a function of the number of posts a user has seen in their current usage session, or the amount of time they have spent on the platform, exit points can be made to be more frequent the longer the user spends on social media.
A function which non linearly increases over time is preferred, ideally one which is bound. This is because with a linear function there will likely be a case when the frequency of exit points based on the function results in all the content becoming exit points instead.

Solutions/Demonstrations/Guide
Good functions to calculate the frequency of exit points would be :

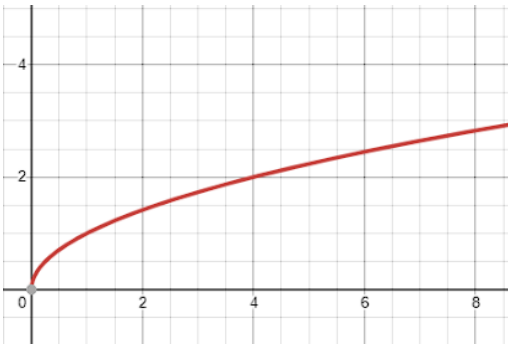
$freq = \alpha log(x) + \beta$



$freq = \alpha e^{x} + \beta$

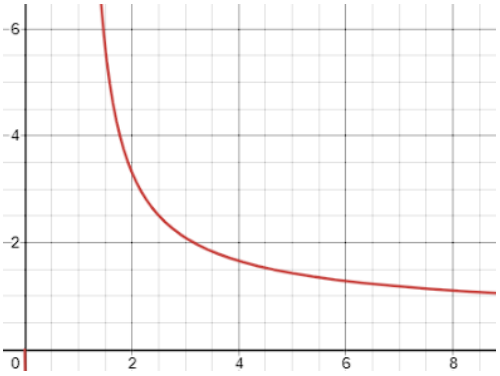


$freq = \alpha x^{1/2} + \beta$

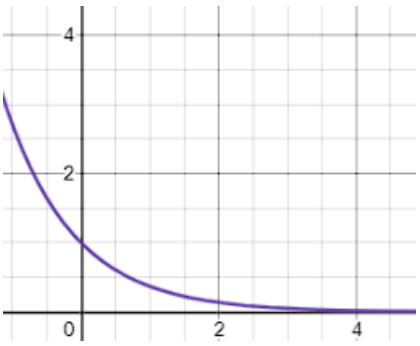


Good functions to calculate the period between exit points would be :

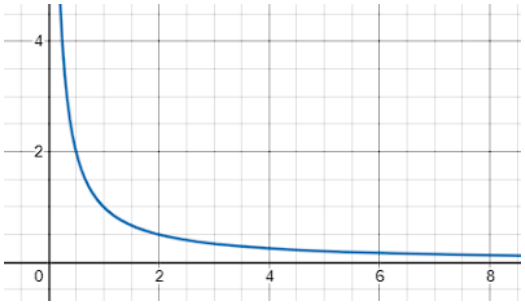
$period = \frac{1}{\alpha log(b)} + \beta$



$period = \frac{1}{e^{\alpha x}} + \beta$



$$period = \alpha\frac{1}{x} + \beta$$



(Images from desmos.com/calculator)

With the parameters α and β being tuneable depending on use case to get the function to behave appropriately for the social media platform it is being implemented for.
Additionally it is recommended to have a maximum frequency of exit point parameters so the final frequency of exit points would be given by $freq = max(f(x), max\ freq)$ with $f(x)$ being whatever function of time or number of posts seen is implemented.
These functions can then be used to determine how many exit points should appear over a set of 100 posts, the number of posts between exit points, the amount of time that should pass between exit points, or any other appropriate usage.
Each function changes at a different rate over time, so which one is the best to use will be platform dependent.
A python notebook containing these functions can be found in the GitHub repository, within the exit point tool folder.

Further Considerations
The type of exit point created is key to the success of the exit point but is very dependent on how content appears to a user within the social media platform. How easy the exit point is to ignore for a user will directly impact the effectiveness of the exit point. With less effective exit points it will likely be required to have a higher occurrence of exit points to compensate for how easily they can be skipped over absentmindedly.
For the exit point to work as intended it is required for a user to have some form of interaction with it to make them think that this would be a good place to end their session on the social media platform.

Time Visualisation and Control

Goal

The aim for this tool is to reduce a user's time spent browsing social media by being transparent with their time spent interacting with specific tagged content, and giving the option to limit their exposure to specific tagged content.

This will be achieved through identifying a simple technical pipeline for this process, and partially implementing the sections of this pipeline relevant to the project's scope.

Ultimately, this tool will reduce the amount of time spent on a platform but increase the quality of time spent, by allowing the user more control over the content they interact with. In turn, this will reduce the likelihood of a user developing a sedentary lifestyle dependent on social media, and in turn improving a user's mental health.

Glossary

- ***Content*** - Any media on a platform that a user can interact with, such as videos and images.
- ***Content Tags*** - A short category descriptor of a piece of content, describing what the content contains. A single piece of content may have multiple tags.

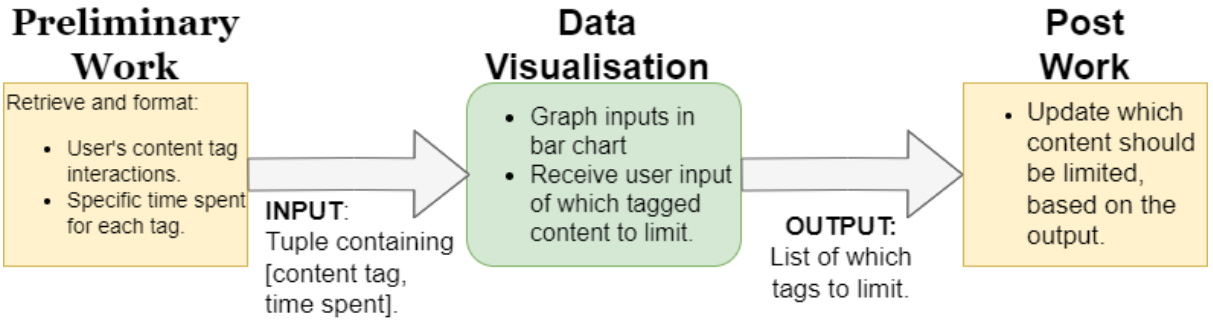
Use Case

This tool requires a platform where users interact with and browse tagged content of any kind, and a platform where users may wish to be aware of, and control, their time spent being exposed to types of tagged content.

The tool can be utilised by a platform wanting to respect their user's access to controlling their time spent interacting with their platform's content, and specifically to a platform wanting to shift a focus from increasing a user's quantity of interactions to quality of interactions.

Technical Overview

A simplified breakdown of what this tool aims to achieve on a technical level is presented as a data flow diagram:



This tool's solution assumes access to the following user data:

- The tags of the content the user interacts with.
- The time spent (in seconds) interacting with this content.

Content can be tagged in different ways. If a platform's content is uploaded by the user base, users may be able to tag their uploaded content manually. Alternatively, the platform itself may provide tagging for content manually, or automatically through feature detection algorithms and classification learning.

Using this data, the tool will communicate to the user their time spent interacting with each type of tagged content (not displaying content they do not interact with), and additionally giving the user the option to limit interactions with certain tagged data. This tool aims to give users more control in their browsing experience, by being transparent and helpful with communicating their browsing habits.

From the tool's output of being able to limit tagged content access, it is also assumed that the platform will be able to prevent a user's access to certain content. This implementation is generalised as an idea in this tool, and discussed further in the pseudocode section.

In summary, this tool will provide a technical implementation for handling time and content tag data, and outputting the visualisation of this data and a user's choice of which content tag to limit. The tool will only discuss pseudo solutions for gathering inputs and reacting to outputs, as on a technical level these implementations are beyond the project's scope.

Solutions/Demonstrations/Guide
To demonstrate this tool's potential implementation, different areas of the proposed solution are demonstrated and explained within the Github, in the Time Visualisation folder

Further Considerations
There are a few considerations for extending this tool's utility.

A major extension would be to decide the time scale of saving the time recordings, which would be specific to a platform's context. For example, an implementation of this tool could reset the recorded times per week or month, to be up to date with the user's interests. For example, would a platform want to maintain the viewing data from years prior to the visualisation, as this may not be relevant to the user anymore?

A further consideration is the idea of giving a user even more power in their viewing habits, specifically allowing them to not only choose which content to reduce their exposure to, but allow them to choose which content they wish to be exposed *more* to. This further gives transparency and honesty to the user, by allowing them to be in absolute control of the content they are recommended. This idea may infringe upon many fundamental design features within an existing social media platform, however, it is a consideration if a platform's context could benefit from this additional transparency.

Providing Effective Moderation

Moderation is a fundamental feature within any platform which encourages user-based content. The concept of moderation is the act of ensuring that user-based content is user friendly to a specific degree, and having an effective way to perform this moderation is crucial in ensuring that users of a social media platform are able to interact with each other in a healthy way.

Poor moderation of a platform can allow for toxic users to directly harass, bully and hate other users sharing the platform, and in some cases a total lack of moderation can allow for extreme platforms to exist which provide a safe haven for sharing harmful, extreme ideologies such as anti-LGBT and anti-semitic views. It can be imagined how individuals being exposed to this kind of extreme content can cause damage to their mental health, especially in the case of targeted bullying. Hence, by considering effective moderation this can help defend and protect users from being exposed to this type of harmful user-based content.

Moderation can be performed manually by humans, which is ideal as humans are generally reliable at labelling text as offensive, however, the human resources required to perform human moderation can be slow and is difficult to scale effectively with a large user base. Hence, an automatic moderation system may solve this issue which requires minimal human validation, which is the basis of this tool.

Model Construction

Goal

Human moderation is currently the gold standard for moderation, but this is often not a scalable solution for social media platforms due to the amount of new content that is being created constantly. As a result there must be an effective way to ensure that this new content is moderated in a scalable way and have posts blocked from being posted or flagged for manual investigation. Natural language processing techniques can be utilised to create a machine learning model that is able to filter out content that is deemed unacceptable for the platform ensuring that the general user of the social media platform is able to avoid seeing content that would be disturbing or unpleasant.

This tool delivers an adaptable implementation of constructing and visualising a generic moderation model, where the model can be adapted for any platform context of labelled comments, basically where the input dataset can be changed for any purpose.

Dataset

The dataset used to create the moderation model was made by the University of Berkeley and is freely available on Hugging Faces[12]. The dataset contains a large corpus of text posts which have been given a hate speech score based on factors such as violence, disrespect and dehumanisation within the post, scored by over 7000 annotators. The dataset contains 39565 posts labelled by 7912 annotators.

[12] Kennedy, Chris J and Bacon, Geoff and Sahn, Alexander and von Vacano, Claudia. *Constructing interval variables via faceted Rasch measurement and multitask deep learning: a hate speech application.* arXiv preprint arXiv:2009.10277. 2020.

This dataset is sufficiently large to produce a good moderation model. It is also a dataset created by a large number of people, this should reduce the chances of any model being trained on this model to inherent biases from creators of the dataset due to the large number of annotators used to build the dataset. This is crucial to ensure the moderation model itself does not allow hate speech against certain groups of people, due to an annotators' biases against that group.
This dataset is easy to use, being easy to download and use straight away with little overhead required, additionally the conversion from a hate speech score to a categorical label can easily be modified to change the sensitivity of the model or add intermediate labels such as a needs human attention label.
Other datasets could be used, and this could improve the performance of the model on a specific platform if the dataset is made up of posts from that platform. However, it is highly recommended to have a diverse team of annotators to try and eliminate any biases being inherited into the dataset.

Modelling

The moderation model is built by using the fasttext library[13] to learn word embeddings specific to the dataset it is given and then use these embeddings to create a model which tries to classify any given sentence/post it is given into an acceptable and an unacceptable category.
The fasttext model once trained can classify posts quickly and can be made to be easily stored on mobile devices, this allows this model to be used fairly robustly.
Fasttext was seen to perform better than alternative natural language processing methods to classify if a post would be seen as offensive or not using the same dataset. Additionally it is able to create word embedding for words outside its training vocabulary such as misspelt words.
Fasttext can be used to only create word embeddings, these embeddings can then be used within a custom model such as a neural network. Fasttext is able to create text representation that performs better than a typical word2vec representation, and performs better than other common text representation methods.
To use any dataset with the fasttext library the posts passed must in the form of a .txt file, with a new post on each line and the label appearing for each post within the same line with '__label__' appended to the front of the label to indicate to the library that that is a label. This may result in some need to edit and reformat a dataset to make it work with fasttext.

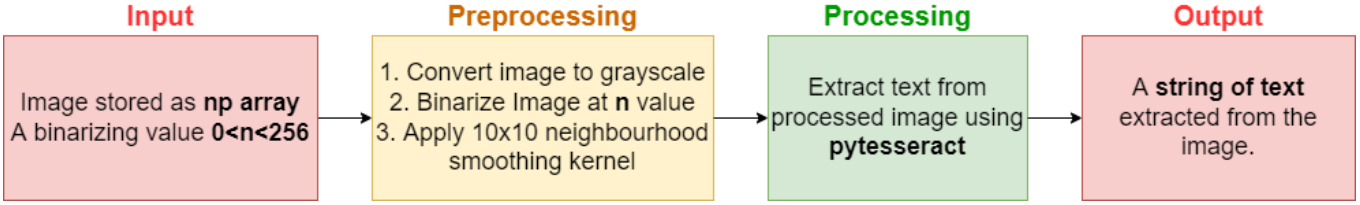
Image to Text

User based image sharing is a common feature of modern social media platforms, and hence is subject to the benefits of moderation. The text-based moderation model can be extended to accommodate for image media where text is present somewhere on the image, as Python libraries exist which can deduce text from a given image. Specifically, the *pytesseract* library[14] which is able to process text within a given image, which would be useful in this case for taking the text from an image and predicting its acceptability label to successfully predict if an image is appropriate or not.

---

[13] Joulin, Armand and Grave, Edouard and Bojanowski, Piotr and Mikolov, Tomas. *Bag of Tricks for Efficient Text Classification.* arXiv preprint arXiv:1607.01759. 2016.
[14] https://pypi.org/project/pytesseract/

The overview of this process is visualised below:



The *pytesseract* library's ability to deduce text is not immediately reliable, as some colours and backgrounds may make it difficult to immediately discern text. Hence, a stage of preprocessing the image using the *Pillow* library[15] must take place to filter out the 'non text elements' of an image, to make text easier to discern. On a technical level, this preprocessing includes:

1. Converting the image to grayscale, as colour information is redundant when trying to find text.
2. Binarizing the image, as the colour of text is likely to be the same or similar colour, so this step separates the text better.
3. Applying a mean smoothing kernel filter, which essentially just smoothens the edges of shapes, and makes text more 'text' like after binarization and erases unwanted noise.

These preprocessing steps, as well as the entire process from an image input to a text output is demonstrated with an example that can be found on the Github Repository within the Moderation tool folder

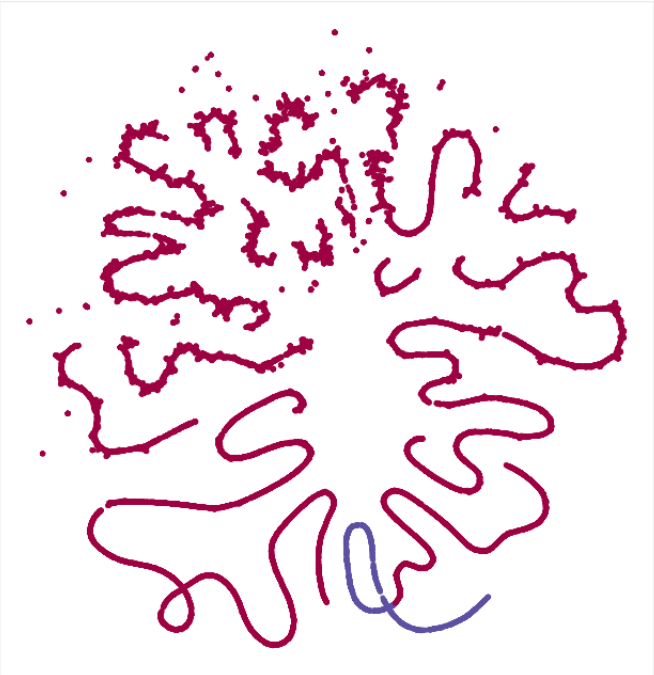
This image to text function is limited to images that may have sensitive text on them that requires moderating, and hence would be redundant against images that display graphic imagery. To solve this, an image moderation model could be produced similar to the text moderation model, through training on graphic and non-graphic labelled images.

Regarding portability, the *tesseract-ocr* engine[16] is available for usage on a variety of languages and platforms, and is hence not limited to this Python demonstration.

Visualisations

A visualisation of the words in the dataset has been made to give an understanding of how the model performs. It reduces word vectors with 300 dimensions down to a two dimensional space by making use of a technique called Uniform Manifold Approximation & Projection (UMAP[17]).

[15] https://pillow.readthedocs.io/en/stable/
[16] https://github.com/tesseract-ocr/
[17] McInnes, L., Healy, J., Saul, N. and Großberger, L., 2018. UMAP: Uniform Manifold Approximation and Projection. *Journal of Open Source Software*, 3(29), p.861.



In this image every red point is an acceptable word in the dataset and every blue word is unacceptable, based on the training of the model. You can see that all the blue words are all adjacent to one another showing that they have a similar vector embedding. Posts which contain these words are likely to be flagged as unacceptable while those made of only red words are acceptable. You can see in the strand of points containing the blue words it is connected to red words, the boundary between these is made up of words which could be unacceptable or acceptable depending on the context they are used within, so could be deemed unacceptable within the context of a post but by themselves are allowed.

The process of producing these interactable visualisations is demonstrated in the Github within the Moderation Tool folder

Further Considerations

There are many further extensions to consider from this moderation tool.

Firstly, there could be considerations into the order of words from the dataset as currently the tool uses the ‘bag of words’ approach, and does not consider their order. Training on the order of words would produce a much more complex model, though many more important features could be extracted from this to produce a more accurate model.

When applying this moderation model within a context, the repercussions of detecting that a user has sent an unacceptable message must be decided. For example, perhaps the message is deleted and the user that sent the message is warned or even educated as to the negative language they have used. These outcomes have not been technically considered within this tool as it is sensitive to the context the model is used within, however it would need to be considered when implementing this tool to render its usage effective.

The Python code can be edited to construct a model from any labelled data as long as it is in the correct format, and so if a different dataset is available it may be beneficial for both identifying platform specific negative language, and if the annotating is done to the standards

of the platform's tolerance, then the model can be more relevant to the goals of the platform, rather than a very generic hate speech level.

The image to text feature could be extended as its own personal project, as this tool was limited to just text based interactions. Potentially, a convolutional neural network model could be trained with *image* data to produce a similar model to this tool but for rendering image data unacceptable rather than text on an image.