



Flee-Mail | Mass Email Detector

E-Business Project - Team 2

Background & Problem Description

Every day, you're bombarded with emails from people who want a piece of you and your time. But are they genuine? Maybe you're a college professor, the founder of a start-up or a company recruiter. Wouldn't it be great if you could tell which emails you receive are genuine, and which ones have been sent to hundreds of other contacts, and not just you personally?

This is a common problem faced by many around the globe. Typically most modern email clients are good at filtering spam and unwanted marketing emails, however they often lack the capability of discerning which emails are sent with genuine intent. It's very difficult and generally not always possible to detect whether an email has been sent to multiple others from only reading its email body. Hence, a tool is required to allow users to 'test' an email for its genuineness.

The Goal

The goal of the project was to provide people with a simple and easy solution to detect mass emails and to determine if the sender of an email has sent the same email to others.

The Solution

The Non-Technical Explanation

Our product Flee-Mail, is a simple yet beautiful tool that fits in snugly to your Gmail inbox. Simply install our add-on, and upon receiving an email, click on the icon to validate whether the email is genuine or not. That's it! On a surface level, it works by checking the body of the currently opened email with a database of other emails sent from the same sender. If the email body matches the content of any other email to a certain threshold, it will classify it as a mass email.

The basis of our project requires many users to have this tool installed in their email client. However, with enough users, it would become an extremely effective tool in

filtering out mass email messages. For example, if it became widely enough used by professors across universities, it would be effortless to detect students who send out mass emails looking for PhD or Masters opportunities without having a genuine interest in the college or professor.

We opted for a manual check approach of mass emails. This is because not every single email that arrives in your inbox needs to be checked. The tool should be used when the user is suspicious of an email or if the email is of a nature in which it could be a mass email – e.g. PhD applications, property viewing requests, etc. If our add-on checked automatically for each email, it would result in a LOT of junk data. For example, simple emails sent out to all the employees in a workplace by management would be flagged as a mass email when opened, which is not what we want.

How to install our tool:

Please note, the add-on is in development stage and not fully published. Only emails under the tcd.ie domain can install the plugin for now.

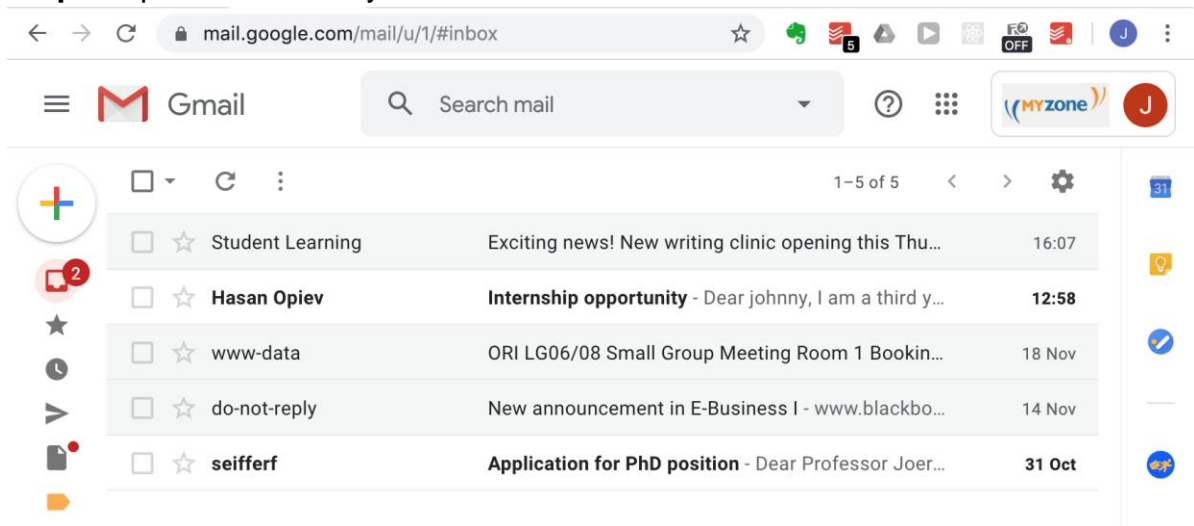
<https://practical-roentgen-6e9cc1.netlify.com/howtocheck>

Video Demonstration:

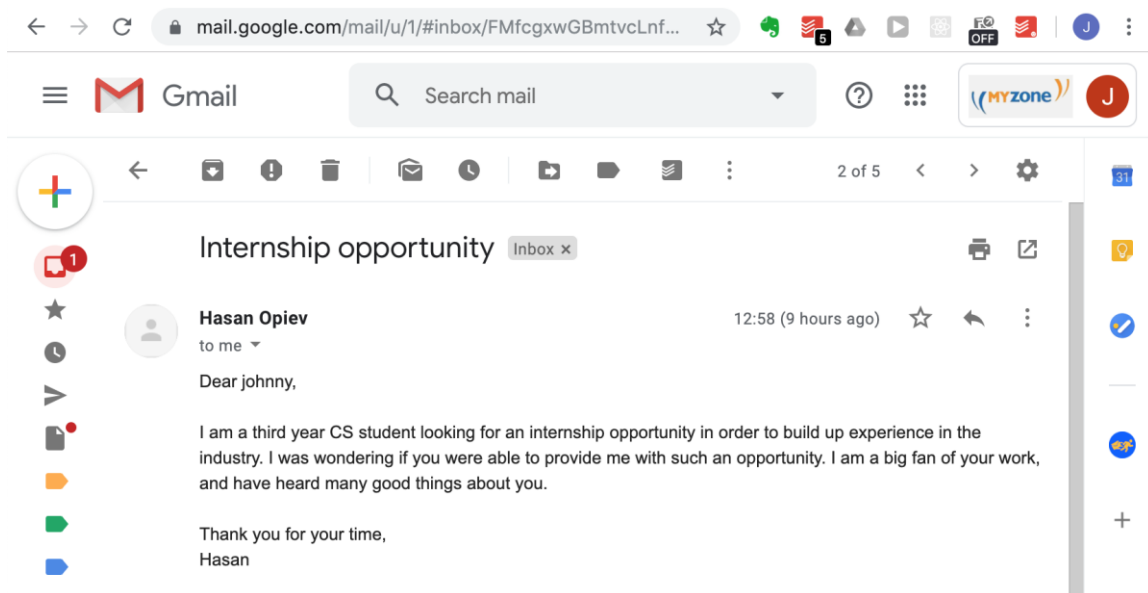
<https://youtu.be/AbEWH2nc4wE>

How to use our tool:

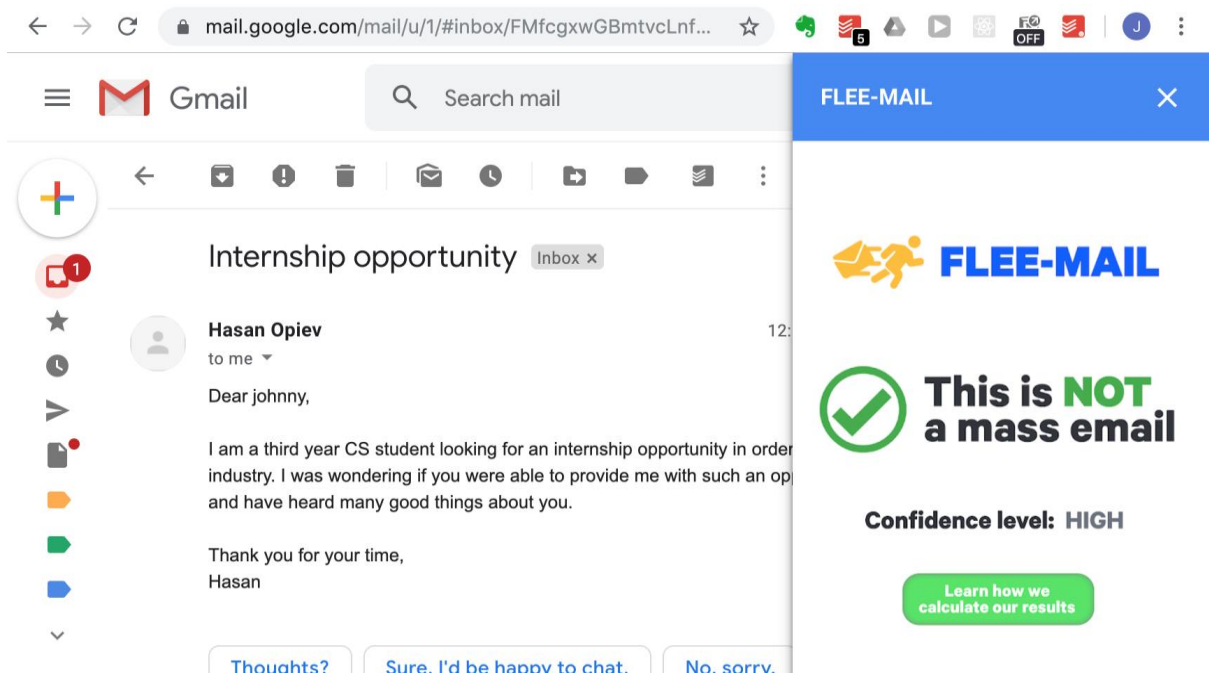
Step 1: open an email in your Gmail inbox.



Here's an email from a student seeking an internship opportunity.

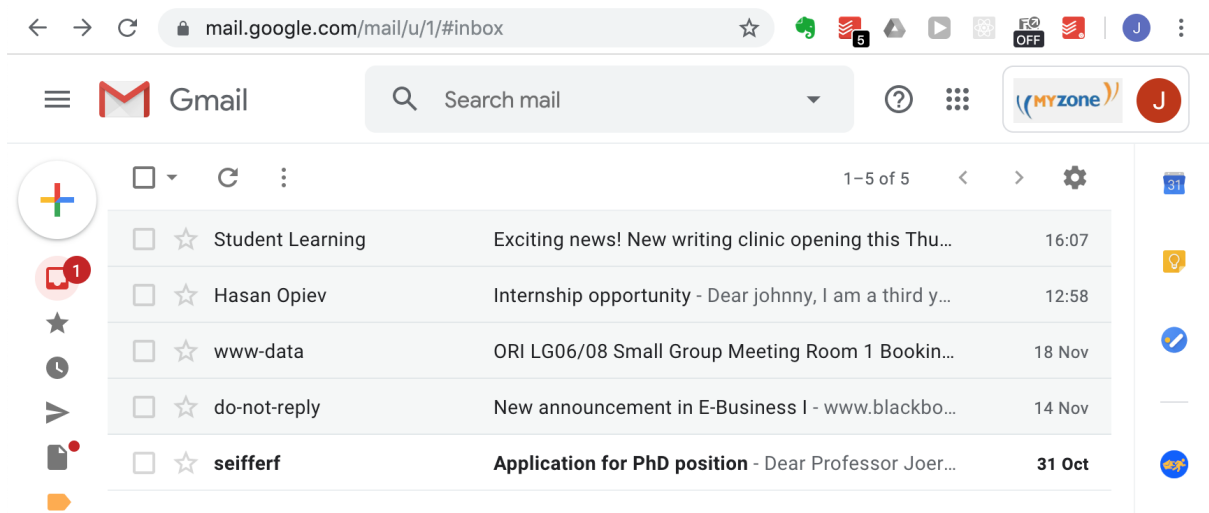


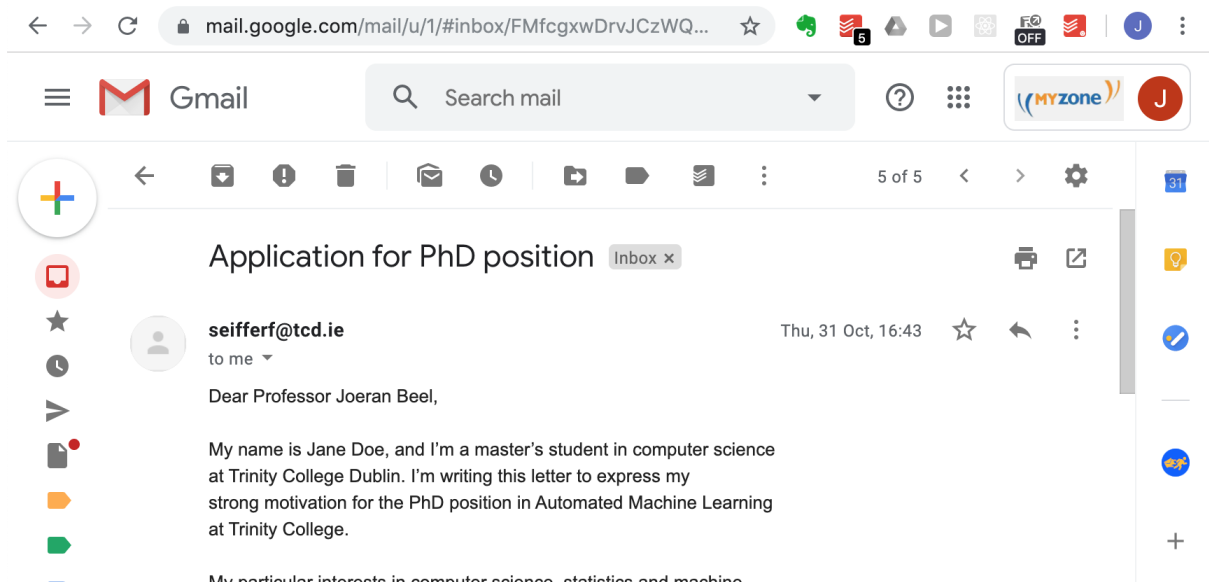
Step 2: Select the Flee-Mail icon in the sidebar to activate the tool. It will automatically check the selected email and interact with our server, returning a result immediately.



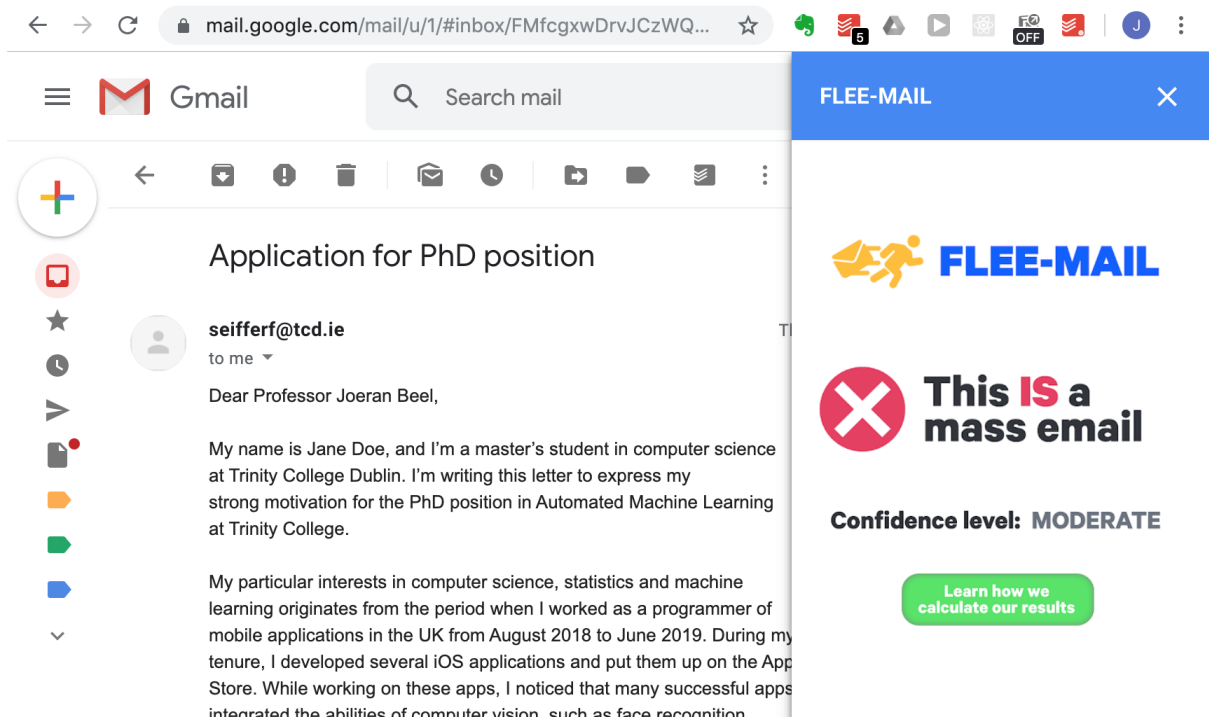
Here we can see that this email appears genuine, and our tool has reported that it is not a mass email.

Here's an example of the tool being used with a mass email which has been sent multiple times to different receivers.





This email is a substantial length. It's unlikely that this student typed up this entire email solely for this professor. Let's check it with our tool:



As you can probably guess, our tool has reported it as a mass email. It has been received by other professors who have also checked it with our tool, registering it on our system.

The Solution

The Technical Explanation

Frontend

The frontend of this Gmail add-on is built using Google App Suite Developer. We chose this because after doing research, it appears to be the best and most straightforward way to integrate an add-on with Gmail. It also allows us to download the add-on to our individual emails without having to publish it.

The programming is completed in Google App Scripts. Google App Scripts operates similar to Google Drive where the project can be shared with multiple users and multiple people can edit and make changes to it. It uses a Google scripting language which is very similar to JavaScript. Google App Suite Developer provides references, guides and samples of features and functions of add-ons which is hugely beneficial as there are little resources elsewhere online about developing Gmail add-ons.

To integrate the frontend and the backend, the frontend sends a fetch request to our database hosted on Heroku to retrieve the information on the email. We check:

- 1) The number of emails that are compared**
- 2) The number of similar emails (matches) found**

The number of emails that are compared determines the confidence level of the result, i.e if only 1-5 emails were checked, the confidence level is moderate. If the number of matches found are 1 or more, we deem the email to be a mass email, and display the result. The result displayed is a simple green tick or red cross, with the appropriate message. We use icons and small messages to make the response of the add-on as clear and concise as possible to the user. As Google App Script does not accommodate the integration of HTML, the best way to provide a stylised and attractive display was to display the results as images, rather than text. We also display an error message, in case the server is under maintenance or has crashed.

The UI of the add-on consists of buttons and widgets. The first button reveals whether or not the email is a mass email. The second button brings the user to the static website for our add-on. We used the bootstrap 4 CSS framework and some basic HTML. On the website, we explain our understanding of a 'mass email', how we calculated whether or not your email is a mass email, and some information on how to install the add-on. The aim of this website is to act as a 'landing page' for people who would like to use and learn about our tool.

Backend & Comparison Method

Comparing Emails

One major problem we had to solve was how to render our understanding of *the same email* in terms that could be computed algorithmically. It is easy to simply compare two texts for strict equality — a problem that could easily be solved by calculating some hash from both texts and comparing the two hashes, or, relying on an even easier approach, by simply walking through the two strings and seeing if we encounter any differing characters before we reach the end of both strings. It is harder, however, to assess if two emails that aren't exactly the same still contain *pretty much the same content* or not. The same problem is also faced in two other areas: Search Engines and Spell Checking.

The goal of Search Engines is to find documents that contain the same content a user entered as a query, but they're obviously not meant to just echo the search query back to the user. Rather, Search Engines usually convert the query into a bag-of-words model that, given a vocabulary, can be rendered as a vector. This vector is then compared to vectors that have been created for the documents in order to calculate a similarity metric that ultimately allows us to return the most similar documents.

Similarly, Spell Checkers tend to go beyond simply comparing words against some dictionary and pointing out words that weren't found. While this is certainly a part of their functionality, they usually also try to find existing words that are similar to what users entered and make suggestions as to what they might have meant to spell. Spell Checkers thus rely on some metric of *similar words*. One approach to calculating this similarity is the Levenshtein Distance Algorithm, which provides an automated way of finding the minimum number of characters that need to be changed to convert one word — or any sort of string — into another.

While both vector space models and Levenshtein Distance would allow us to assess the similarity of two emails, rather than just their complete equality, the Levenshtein Distance Algorithm has a particular advantage. In order to obtain a vector space model, emails would first need to be converted into bag-of-words models. This, however, would mean that we discard all, or at least most¹ information about the ordering of words. Since our goal is to assess if one email could be a slightly modified version of another one, rather than to assess if two emails contain similar vocabulary, we concluded that Levenshtein Distance would be a better fit for our application.

The Levenshtein Distance Algorithm allows specifying the costs of inserting, deleting and modifying some element. We decided to use a fairly standard configuration, where we set both the cost of insertion and deletion to 1 and the cost of modification to 2, since replacing an element with another one can be considered equivalent to deleting the first element and inserting the second.

Furthermore, we decided to calculate Levenshtein Distance on lists of tokens,² rather than using the more common approach of calculating it on strings (lists of characters). Using some sort of tokenisation (i. e. splitting text into tokens) made sense in our case, since intuitively, it makes more sense to calculate the number of differing words rather than the number of differing characters.³ Since we wanted to

keep our tokenising algorithm as slim and straightforward as possible, we decided to simply use a regular expression to insert a space in front of any number of known punctuation marks and to use another one to split the text of an email into different parts wherever a continuous region of whitespace occurs.⁴

In order to assess the difference between two emails, we first calculate the Levenshtein Distance between them and divide the result by 2, which is the cost we set for the replace operation. We then divide this absolute difference by the length of the email in question⁵ to get the relative difference. Finally, we subtract this relative difference from 1 to get a measure of overlap between two emails. We thus arrive at the following formula for calculating the overlap between an email m and another email m' :⁶

$$\text{overlap} = 1 - \frac{\text{levenshtein_distance}(m, m')}{2 \cdot \text{length}(m)}$$

Given this measure of overlap, deciding whether two emails contain *pretty much the same content* or not comes down to finding a threshold above which they could be considered the same and below which they could be considered different.⁷ Deciding which threshold to use obviously requires tweaking the value based on some manual assessment of how well the system performs on a set of example emails. While we didn't do any extensive testing, preliminary experiments suggest that 90 % might be a good threshold value, which is why we decided to use this value for now.

Backend

As explained in the previous section, we decided to use the Levenshtein Distance Algorithm to measure the overlap between emails, which serves as the basis for deciding whether we consider other emails in our database to be the same as some email in question or not. This information can then be aggregated to inform users about how many other people have received the same email as the one that is being checked.

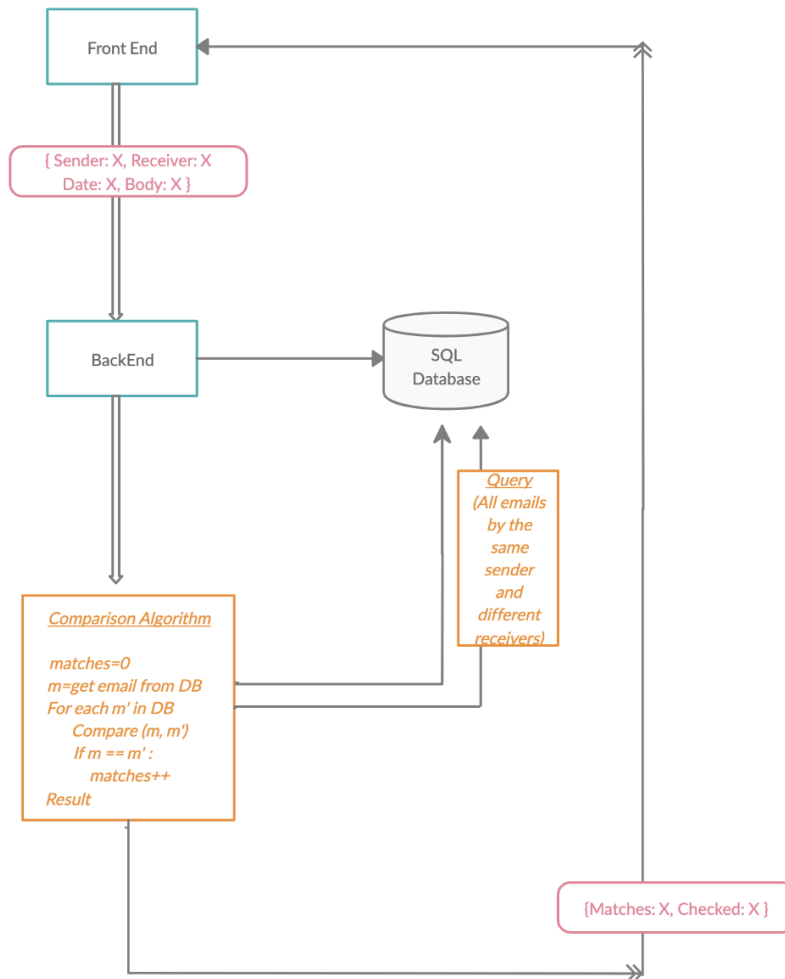


Figure 1: Diagram showing the architecture of our application

For doing these calculations, the frontend transfers an email's sender, receiver, timestamp and text body to the server,⁸ where it is processed. The backend processing is outlined in figure 1.9 First, the backend stores the email in an SQL database, using the combination of sender, receiver and timestamp as the primary key. By using this information as the primary key, we also ensure the de-duplication of entries. De-duplication is important, since the same user might query our backend with the same email multiple times simply by clicking the add-on's icon on multiple occasions. Obviously, though, we would not want to consider previous queries by the same user as evidence that some email is Mass Email. After storing the email in question, the backend executes our comparison algorithm, which retrieves the email (m) as well as any other email which has the same sender but a different receiver (m') from the database. The comparison algorithm goes on to compare the email in question to any other email (m') and accumulates a count of how many of these other emails are the same as the email in question, using the similarity metric described above. Finally, the comparison algorithm returns the number of emails that matched the email in question to the frontend.

If our database doesn't contain any other emails from the same sender, however, we can't really tell if the email in question is Mass Email or not, since we have no data to

compare it to. If, on the other hand, it contains lots of emails from the same sender, we can be fairly certain that our results are informative. Therefore, the comparison algorithm also returns the total number of comparisons that have been executed to the frontend, where this information is used to display a confidence score.

Limitations & Outlook

In retrospect, we were pleased with many aspects of our development process. Developing our backend with Node.js, Express.js and MySQL allowed us to get an early prototype up and working quite quickly. We connected our Github repo to a server hosted by Heroku with automatic deployment once a commit was made, which was extremely helpful when testing our backend with the Gmail add-on.

Using the Google Apps Script technology was far from convenient. It's extremely limited, has poor online support and very little resources on online forums. Their documentation wasn't straight forward, and it took quite a lot of effort to even create a button that opens a web page when clicked. Our add-on is limited in the fact that it can only read and check email messages with plain body text. Multimedia and HTML formatted emails don't work with our plugin.

The backend is currently still in a very early stage of development and could certainly still be improved in terms of performance and robustness. We do, for instance, calculate the whole Levenshtein Distance between any two emails we compare, although we actually just need it to check whether it is below a certain threshold or not. The performance of the Levenshtein Distance Algorithm could therefore be improved by skipping those parts of the calculation that are already known to lead to results that will exceed the threshold.



Furthermore, we are currently using a monolithic architecture for our application. If we were to scale it, it would probably be a good idea to split the Comparison Algorithm out into a microservice that can be scaled individually. Since we already separated concerns when designing our architecture, however, this should be a rather easy and straightforward thing to do. Additionally, further experiments are needed to find the optimal threshold value for deciding whether emails are pretty much the same or not.

In conclusion, we were delighted with our end product and believe that it sufficiently solves the problem of mass email detection. We enjoyed working together as a team and learning from each other's skillsets. We hope this tool is useful to you and we are open to critical feedback on how to improve our work.

Footnotes

1. It would certainly be possible to use ngrams of various sizes, rather than just the words themselves. As we increase the ngram size, however, we move closer to simply comparing string equality, and the smaller the ngrams are, the less information about word order is preserved. ➡
2. Token is pretty much a technical term for saying word. Since there is lots of discussion in linguistics about what exactly a word is, however, it is common to use the term token to denote whatever small parts a program happens to split a text into. ➡
3. If we just considered characters, the Levenshtein Distance between the words “house” and “mouse” would be 2, since one just needs to remove one character and add another one. The difference between “house” and “mansion”, on the other hand, would be 10, since one has to remove 4 characters (“h”, “o”, “u” and “e”) and add 6 new ones (“m”, “a”, “n”, “i”, “o” and “n”) — actually, one might also remove “h”, “u”, “s” and “e” and add “m”, “a”, “n”, “s”, “i” and “n”, but that would still be 4 deletions and 6 insertions, so it wouldn’t make a difference. While this sort of reasoning is great when dealing with spelling errors, it doesn’t make that much sense for comparing the content of emails. As far as we are concerned, replacing “house” with “mouse” changes some given text just as much as replacing it with “mansion”. One might even argue that “mansion” is supposed to be closer to “house” than “mouse” is, but this sort of argument is probably more of an exercise in philosophy than something that could actually benefit our implementation. ➡
4. While one might argue that a more advanced approach could yield results that are more in line with linguistic theories, we concluded that this simple approach should be able to serve as a solid basis for our program. Furthermore, we strongly suspect that using more accurate tokenisation techniques would have no impact on overall performance. ➡
5. We could also have used the length of the other email or the average of the two. We chose to simply use the length of the email in question, though, since the length of the other email changes for every comparison, which makes the implementation slightly less straightforward. It also makes sense intuitively to set the email in question to be the scale things are compared to. ➡
6. As explained above, we decided to tokenise emails before processing them. Both `levenshtein_distance` and `length` are thus calculated on lists of tokens, rather than on strings. ➡
7. In fact, given some fixed threshold, the above formula can be solved for `levenshtein_distance(m, m')` in order to express the threshold in terms of the maximum Levenshtein Distance we would still consider to be indicating that two emails are pretty much the same. We consider emails to be the same if the overlap is above our threshold. In mathematical terms, this can be rendered as:

$$8. \quad \text{threshold} \leq 1 - \frac{\text{levenshtein_distance}(m, m')}{2 \cdot \text{length}(m)}$$

9. Since only the output of `levenshtein_distance` depends on m' , it makes sense to convert this formula into a form where one side of the comparison can be calculated only once every time an email is checked against our database:
10. $\text{levenshtein_distance}(m, m') \leq 2 \cdot \text{length}(m) \cdot (1 - \text{threshold})$
11. We thus implemented the comparison by calculating the right hand side of the last equation once for every incoming email and checking if the Levenshtein Distance calculated for any pair of emails is less than or equal to the resulting value. 
12. Transferring and storing all this data obviously raises certain security concerns. We addressed these by ensuring that the communications between frontend and backend are properly encrypted and that the database is properly secured, so that only our backend can access it. We also provide users with detailed explanations of what data is transferred to our server and of how it is used, so that they can make an informed decision about which emails they want to check with our service. 
13. We used javascript and Node.js to implement the backend and rely on Heroku as our hosting solution, which provides both a basic Node.js installation and an SQL database for us to use. For calculating the Levenshtein Distance, we used the `edit-distance-js` library. Since that library used to operate on strings, rather than on lists of arbitrary elements, however, we had to directly include a modified fork of it in our code base. Our changes have since been merged into the upstream library, so we should be able to remove that fork and return to using the external library once it reaches its next release. 