Smail - Smart Email Client

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Sahil J. Chaudhari and Vishesh Munjal
(111801054 and 111801055)

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Smail - Smart Email Client**" is a bonafide work of **Sahil J. Chaudhari and Vishesh Munjal** (**Roll No. 111801054 and 111801055**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr. Albert Sunny

Assistant/Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

The BTP project assigned to us is a great opportunity for learning and personal growth. The project started as a vague idea to counter some problems that were generated due to the communication gap in the recent pandemic. Therefore, We are grateful for this chance and are grateful to our professors who have motivated, contributed and modified this vague idea and gave it the shape of a budding project which is yet to bloom.

First of all, we would like to thank our college, IIT Palakkad, for making it possible for us to have such an opportunity exist in the first place. We really appreciate the hard work our professors, along with the academic section put in to tutor us and make us the best version of ourselves.

Second of all, I would really like to thank Dr. Albert Sunny for being the project mentor. Due to Dr. Albert only, we were able to extend the project - smail client with additional features which are unique to the project and may come in handy to anyone using the client. His insight and helpful nature are in some sense the backbone of our project.

We would like to thank Prof. Unnikrishnan C, the BTP project coordinator for his helpful nature and understanding. We really appreciate the effort he puts in to make us understand the relevance of the project as well as the norms to be followed. Also, we are thankful for the regular updates and his patience with the deadlines.

Due to all of the above-mentioned people, directly or indirectly, we hope this experience becomes an unforgettable golden opportunity that will help us be successful in our career.

Abstract

The Smart Email Client is an android application with a custom build AI to categorise and prioritize emails via easily customizable coloured classification according to users organization needs. The Client will be packed with a custom UI and exciting features such as auto google calendar integration , google assistant integration, reminder system with custom notification according to priority, text-to-speech and speech-to-text integration.

The client even extends to more features but essentially to focus on the main feature is the custom priority that a user/organisation experiences with the focus to prioritize important mails and display them accordingly to their priority class and colour code. A follow up feature of this is to have calendar integration to handle notifications for deadlines or events that will be automatically handled by the client. The extended features aim to make the client as user friendly as possible with an advanced option to use google assistant to convert voice to text mails.

Contents

List of Figures

Chapter 1

Introduction

The Smail - Smart Email Client is an android application with a custom build AI to categorise and prioritize emails via easily customizable coloured classification according to users organization needs. This will allow the user to give importance to the mails they are interested in and will make sure that such mails are never misplaced or forgotten like a needle in a haystack. The basic implementation behind this feature is to use labels or in layman's terms certain keywords and a collection of such keywords will generate the priority index for the respective mail which can then be further classified into various categories and color coded. This implementation can then be refined further and further using the mail context to create a smart client as the name suggests. The project after completing its initial hurdles has entered into phase II with the next goal in mind i.e. to finalise the priority algorithm and to integrate the gmail API with the app along with few tweaks in the front-end in order to achieve the promised user friendly environment for the user.

1.1 Motivation

The motivation behind the project came from a vague idea to counter some problems such as missing deadlines conveyed through mails or unread mails due to bombardment of mails

from all sub divisions of college namely clubs, academics, TPO etc. that were generated due to the communication gap in the recent pandemic. All the above mentioned problems lead to a need for a custom priority system for emails which further with some insight from our mentor was extended to other features as well which makes the client useful and handy. Thus the project Smail emerged.

1.2 Extension of Project

The further goals of client is to integrate the priority algorithm with the app and finalise the extended features. Also, in order to assure smooth working of the app we have decided to design a pre-processor for the data (i.e. essentially the mail body) that is supposed to be passed to the algorithm in the back end. Furthermore, the extended features which are subject to time constraints, hence we will try to add as much extended features as feasible. We would add text-to-speech integration as well as speech-to-text integration for user convenience. Another goal is to integrate google assistance with the app. In order to avoid deadlines or important events, the calendar alarm /notification feature will be another end goal of the upcoming half semester. Not to forget date extraction from the data(i.e. mail body) to figure out the essential details about an event.

1.3 Overview of The Report

This chapter dealt with the introduction of Smail client along with motivation behind it and also, goes into the further extension of the project. In the next chapter i.e. chapter 2, we talk about the progress accomplished in the project up-till date in phase II. In chapter 3 and 4, we talk about the previous algorithm and the finalised one. We also compare and explain about the algorithm and data structures used. In chapter 5 we give a brief insight about the future works that are to be accomplished. In chapter 6, we showcase the results of the work done. We will close our report with conclusion and contribution of each

member toward project. And Lastly include all the references that we have used for this project so far.

Chapter 2

Review of Prior Works

This chapter provides a detailed analysis of the progress made in phase II and a brief overview of the tasks accomplished in phase I.

2.1 Phase I overview

Phase I started with careful thorough research and analysing various components such as installing Android Studio on Windows and flutter. We also fixed dart language for designing and display it on device emulator. Due to which we were able to design the Interface of Smail - Smart Email Client. Made important decisions on the architecture. Also, we deal with the Google API's authentication to establish the credentials for use of google API(done in this phase). Lastly but not the least we focused on the backbone of the project i.e. the priority Algorithm, and after multiple refinements we made an acceptable algorithm for the priority handler of the app along with working demos.

2.2 Phase II Progress

2.2.1 Gmail API Study

We read documentation about gmail.v1 library Api given on googleapis dart webpage [1]. We studied required scopes and permissions we need in OAUTH to read emails from gmail and also created map of GmailApi instance. As shown in Fig 2.1, GmailApi() constructor

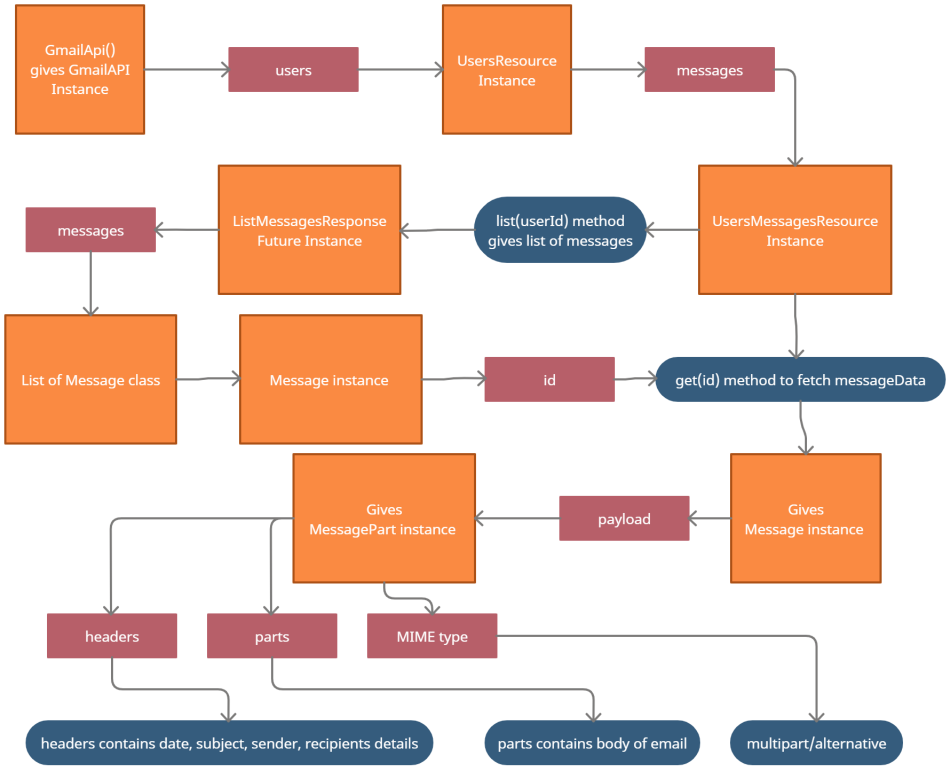

Fig. 2.1 GmailApi Instance.

gives GmailApi instance which have property users which gives us UsersResource instance, which have all information about user in gmail, i.e messages, data, etc. We are using messages property of it which gives UsersMessagesResource instance, which have list method which takes userId parameter i.e emailId of user, which gives ListMessagesResponse future instance which have messages property which gives list of Message class. By iterating each Message instance we will get id of message, using it with get method of UsersMessages-

Resource instance we will get Message instance which have payload property, which stores all information regarding particular mail,it returns MessagePart instance, we will focus on three properties mainly:

- **MIMEtype**[2]: MIMEtype gives us layout in which data is stored, for our project, all emails are in type multipart/alternative, i.e body of email and attachments will be find in parts property of class.

- **parts**: This property gives List of MessagePart instance where in body property of first element, we will get messagePartBody instance which have data property that gives us body of email.

- **headers**: This gives MessagePartHeader which have name and value property, where name is name of header such as sender, time, subject, etc. and value have its respective value.

Using this information we will integrate OAUTH and Gmail Api in our application.

2.2.2 OAUTH Integration

As we have mentioned in Phase 1, we have taken permission by google cloud console to user gmail API in our developement of application, have added SHA key to firebase console under google authorization. We then downloaded json file name google_services.json to our repository, and then called GoogleSignIn class instance to authorize our application to view user's data. But in order to get authorization, we have to mention in scopes of class constructor that we need permission regarding readonly of gmail data of user. Once added, and run, on opening of application when user sign in to app, it will ask permissions to user whether to allow or decline to view their gmail data.

final GoogleSignIn _googleSignIn = GoogleSignIn.standard(scopes: ⟨String⟩[GmailApi.gmailReadonlyScope,],);

This is the way we can initialize OAUTH with scope[3] for readonly of gmailApi using GmailApi's constant gmailReadonlyScope that is constant string value.

var httpClient = (await _googleSignIn.authenticatedClient())!;

Then we will authenticate client using above line of code.

2.2.3 Integration Of Gmail API

We will use flowchart shown in fig 2.1, first we initialized gmailApi instance with httpClient shown in above section.

ListMessagesResponse results = await gmailApi.users.messages.list("me", maxResults:maxResults); using this we will fetch list of messages using list method. Then we will iterate over results to get id of each message and then will call get method as following: **Message messageData = await gmailApi.users.messages.get("me",id);**. Now we will fetch body of email using messageData using following code: **String text = stringToBase64Url.decode(messageData.payload?.parts?.first.body?.data ?? "");**. Using **messageData.payload?.headers?.forEach((element)** we will iterate over each header and will fetch data of sender, date, time, recepient, etc using list of headers **List⟨String⟩ headers = ["Subject", "Delivered-To", "Received", "From", "Date", "To"];** and key and value of property of header and then will store all of it in AppMessage class that will be used to present emails to front end.

2.2.4 Priority Algorithm Finalisation/Modification

Now since we are able to fetch mails, the time has come to finalise the Priority algorithm so that it can be integrated. Our algorithm has undergone many changes which have been thoroughly discussed in the upcoming chapters. We modified the algorithm in such a way so that multiple tags could be supported. This benefits us to get a better heuristic in regards to priority categorization of the emails. Also, we added method to modify tags in order to be fair, i.e. in some sense the algorithm learns on its on via adding keywords in

specific tags. We kept a record of the frequency of the word appearing in a certain type of mail and used the LFU replacement policy to get the new keywords for a particular tag. Moreover, these benefits come at literally minimal cost i.e. the time complexity of the algorithm doesn't even change although these is change in space complexity but that's only temporary as we are using dynamic allocation.

2.2.5 Front End Modification

Aesthetic appeal has a great impact in any project/client nowadays, our project has also never forgot about it and has always tried to improve it. In-fact, the motto of our project is to make our app user friendly and aesthetically appeasing. So we made a few tweaks in our front end using flutter widgets[4] to achieve this, we made our login page for logging into the application with the respective email ID. Since we are supporting multiple tags we made sure that those can be seen on the interface. Moreover, we changed the outlook of how mail used to look, making a bit more nice and rounded. Also we added a config page where the user can visit to change the configuration according to their respective needs. Lastly but not least, we also added a signout option which will take us back to the login main page. All these might be hard to visualize but not to worry the later chapter 6 Results give a visual representation of all these changes.

Chapter 3

Phase I Algorithm

In this chapter, we will dive into the algorithm constructed in phase I. To note that the algorithm of phase I was constructed mainly in 2 steps. We shall give a brief overview of the final Phase I algorithm.

The first algorithm proposed in phase I had 3 components namely, TextParser, GetPriority and SetTag. TextParser which calculates priority and returns priority value along with suitable tag, GetPriority which take batch of emails and parse those email in array of string for subject and body and calls TextParser method and assign most suitable priority value and tag among those received for subject and body, and last is SetTag which creates new tag according to user given values and store it in various global arrays for front end and above methods.

Later on with the help of data structures the algorithm was refined further. Tries were used to parse the content of the email where each word was inserted in a trie tree which in turn makes it faster to search a specific Keyword. We also used unordered maps in order to make search, insert and traversing a bit faster.

Let's have a look at the algorithm designed and tested in phase I.

Algorithm 1 TextParser(DataTree)

```
1: for Tag in taglist.Sorted(High to Low : PriorityNum) do
2:     for keyword in Tag.keywords do
3:         if (DataTree.contains(keyword)) then
4:             count++;
5:         end if
6:     end for
7:     if (count >= Tag.Threshold * Size(DataTree.keywords)) then
8:         return Tag.PriorityNum, Tag.Name;
9:     end if
10: end for
```

As the name suggests, the job of TextParser is simple i.e. to parse the text. It requires a DataTree(a trie tree) as input. Then for tags in the taglist according to high to low priority its checks whether the email has a certain set of keywords. If the email does indeed cross the threshold to belong to that tag, it was then assigned the Priority Number.

Algorithm 2 GetPriority(EmailData[] PooledEmail)

```
1: for Email in PooledEmail : do
2:    TreeNode DataTree = TreeNode();
3:    for word in Email.Subject.words do
4:       DataTree.add(word);
5:    end for
6:    SubjectPNum = TextParser(DataTree);
7:    DataTree.clear();
8:    for word in Email.body.words do
9:       DataTree.add(word);
10:   end for
11:   BodyPNum = TextParser(DataTree);
12:   EmailMap.add(Email.ID, Max(SubjectPNum, BodyPNum))
13:   EmailTag.add(Email.ID, Argmax(SubjectPNum, BodyPNum).Totag)
14: end for
15: return True;
```

GetPriority method takes care of organizing and assigning the priority numbers of all emails. It takes an array of emails and for each email create 2 data trees, one for the subject and other for the body. Then it calls the TextParser on both of these trees separately and assigns the Priority number of subject and body to the respective email ID.

Algorithm 3 SetTag(TagName, str[] keywords, PriorityNum, PriorityColor, Threshold = 0.8)

1: **if** *not exist* : **then**
2: $t = CreateTag(TagName)$;
3: $t.keywords = keywords$;
4: $t.PriorityNum = PriorityNum$;
5: $t.Threshold = Threshold$;
6: $PNumMap.add(PriorityNum,\ PriorityColor)$;
7: $TagMap.add(TagName,\ t)$;
8: $TagList.add(t)$;
9: **end if**

As one of the main objective of our project is to provide user friendly interface where it can change or set its own tags. Also for us to have a predefined set of tags. To kill these 2 birds with one stone, we created the SetTag method to set a tag, it takes in the Tagname, an array of keywords, Tag's desired priority and colour associated with it to create a tag for us which is appended into the Taglist.

3.1 Conclusion

In this chapter, we gave a recap of the algorithm of Phase I which essentially had 3 parts:-

- **TextParser** which currently has a time complexity of $O(ntk)$ where n = length of word , t = number of tags, k = number of keywords.
- **GetPriority** which also has a time complexity of $O(ntk)$.
- **SetTag** which have the time complexity of $O(t)$

The algorithm is further optimized in the next chapter with due consideration so as to not increase its computation otherwise it may make the functioning a bit laggy. The

optimization bring out the best in the algorithm where complexity sure does increase only to make the priority handler better.

Chapter 4

Phase II Algorithm

In previous chapter, we have seen proposed priority algorithm for our smart email client for phase 1 which have time complexity bottled by the GetPriority algortihm and TextParser algorithm by using trie and unordered map i.e. of the $O(n^4)$ to $O(n^3)$. This algorithm only supported one tag per email. In this chapter, we will propose priority algorithm of same maintained time complexity i.e of $O(n^3)$ with enhanced feature optimally and also bring fairness. We have supported multiple tags per email, also used Least frequently used cache method to find top K most frequent word in $O(nlog(K))$ where n is number of words in text and K is Constant and hence in $O(n)$. These will help to enhance Tag's keywords. We also used early stoppage when conditions are met rather than completing entire loop to save several computational power.

4.1 Construction

In this section we will see, how our new proposed modification over previous algorithm is supporting multiple tags and also bringing fairness in finding appropriate tags and keywords for them in same time complexity.

Our first modification is, we have keeping track of frequency of word in particular email and also keeping track of top K most frequent word. We are achieving this by using

unordered map as finding frequency of word in hashmap takes $O(1)$ which will keep track of frequecy of all words in email body as well as in subject and by using least frequent used cache method to find top K most frequent words.

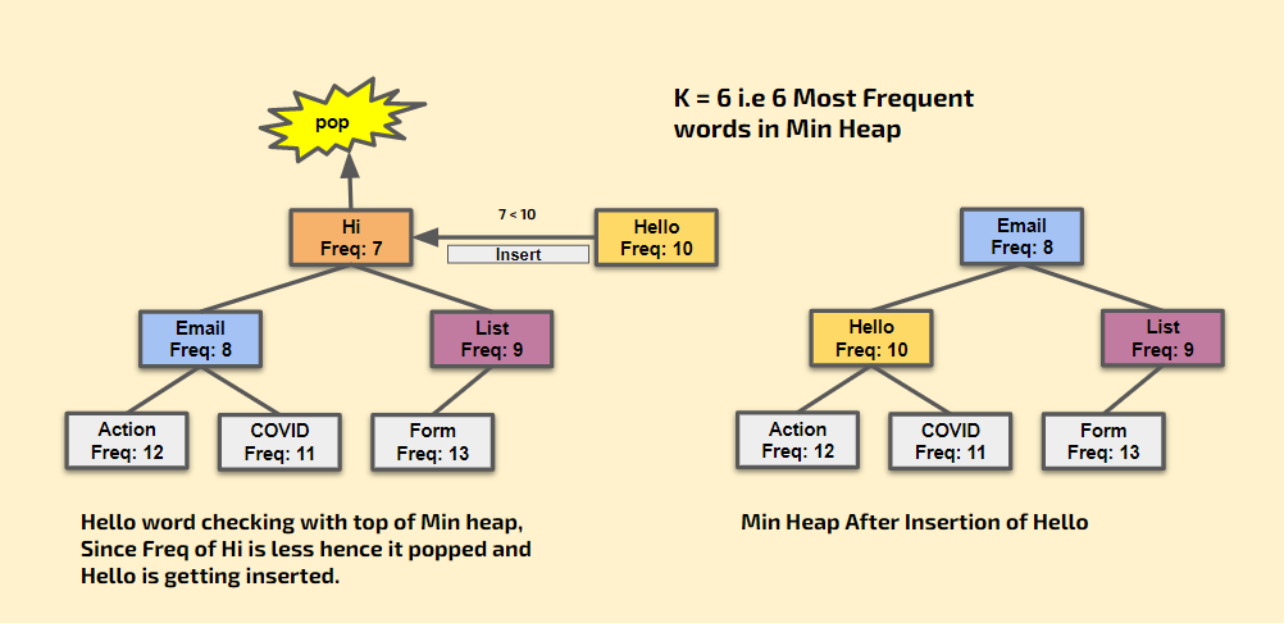

Fig. 4.1 LFU MinHeap Demonstration for K Most frequent words.

Let us understand how we are using LFU cache replacement policy method [5] to find top K most frequent, as in fig 4.1 shown, we are using min-heap to keep track of top K most frequent words at any instance where top node will always have word with least frequency in K most frequent words, let say new word hello is coming then we will check frequency of hello word in frequency hashmap and will check whether frequency of top of minheap is less than frequncy of new word i.e hello, in this case it is smaller and hence we will pop top element and insert hello node with frequency value in minheap. Insertion and deletion of minheap both takes $O(log(n))$ but in our case n is K i.e constant hence maintenance of K most frequent word will be done in constant time for each new word arrival and hence to check for all unique words of email we will require $O(n)$ time where n is number of words.

Now question arises, why we need K most frequent words, answer is to modify list of keywords for tag. As shown is fig 4.2, let us consider that our K is 3 and we got three most

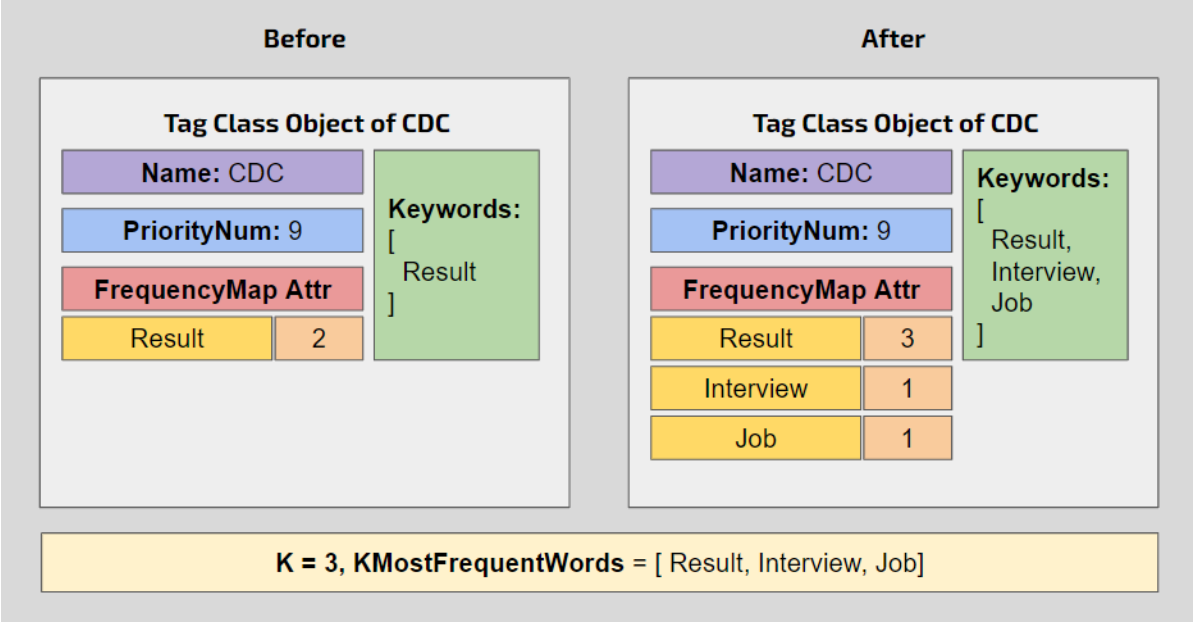

Fig. 4.2 Modification of Tag object using K Most Frequent words.

frequent words from email's words from both subject and body, now one word i.e result is already in our tag CDC's keyword list but words interview and job are not in it. So we will insert these two as well in keyword list, but if we consider this expanded list of keywords to find optimal tags, every time computation is increasing and hence we have to restrict it.

Restriction is done by frequencyMap attribute of Tag class which have described in coming section. This attribute keep track of number of time each keyword in modified list in tag has appeared in K most frequent words of every email, if it is in those words then we will increament frequency of that word by 1 and if new word is inserted in keyword list then it will be initialized as 1. So in above fig, result was there for 2 time before and hence after evaluation of current mail, it's frequency went to 3 and frequency of job and interview initialized to 1. Now for upcoming email, will choose top K' most frequent keywords from keyword list according to hashmap, and those will participate in evaluation of tags. This way for each email, for each tag we will only consider top K' most frequent keywords from list and hence time complexity will not be affected and remain same for every email's evaluation.

To find multiple tags for each emails, Instead of returning one tag when threshold satisfied, we will keep finding tags that are satisfying threshold and store them in list and when there are tags of desired value, let's say two i.e max two tags per email will be calculated and function will return that list. Since in previous chapter's algorithm, we anyway traversing through each tag for evaluation, time complexity wont be affected, just we will require additional list to keep track of those tags. We will also break loops when following conditions are met:

- If keyword threshold satisfied then we will break keyword loop in TextParser algorithm.
- If tag counter i.e number of tags suited for emails reach limit max allowed tags per email then return list of tags.
- If keyword count reach limit i.e top K element from keyword list of tag undergoes evaluation then break keyword loop.

4.2 Improved Method

As most of the structure and parameter names are similar to that of algorithm proposed in chapter 3, we will only discuss changes in following algorithms according to explanation given in section Construction.The SetTag method will remain unchanged.

Algorithm 4 Class Tag

```
1: string name,
2: string[] keywords,
3: float PriorityNum,
4: Unordered_map frequencyMap,
5: float Threshold,
6: Tag(name) {
7:         self.name = name
8:         self.keywords = []
9:         self.frequencyMap = new Unordered_map(string,int)
10:        self.Threshold = 0.2
11:        self.PriorityNum = 0
   }
```

Above is a Tag class, which has been changed due to various requirement by other methods. This class will have following attributes:

- **name**: This will be name of tag given by default or by user
- **keywords**: This will be list of words that will be used to identify appropriate tags for email
- **frequencyMap**: This will be hashtable i.e unordered map where key will be keywords and value will be number of times they are appeared in unique emails.
- **PriorityNum**: This will be float value in range 0 to 10, depending on importance of tag where 10 is highest importance and 0 is least importance.
- **Threshold**: This will be float value from 0 to 1.0, which will decide minimum fraction of keywords should be present in email text to assign tag to that email.

Constructor of Tag class will take name of tag given as parameter and will assign it to name attribute, priorityNum attribute will be initialized as 0 and Threshold attribute as 0.2 and both keywords and frequencyMap will be initialized as empty.

Algorithm 5 TextParser(DataTree)

```
1: TagCounter = 0
2: TagMatched = []
3: for Tag in taglist do
4:    presentCount = 0
5:    keywordCount = 0
6:    for Keyword in Tag.keywords do
7:       if DataTree.contains(keyword) then
8:          presentCount++
9:       end if
10:      keywordCount++
         // break loop if Threshold satisfied
11:      if presentCount ≥ Tag.Threshold*(Size(DataTree.keywords) then
12:         TagMatched.add(Tag)
13:         TagCounter++
14:         if TagCounter ≥ MAXTAGSALLOWED then
15:            return TagMatched
16:         end if
17:         break Keyword loop
18:      end if
         // if top K Keywords are tested then break keyword loop
19:      if KeywordCount ≥ topKKeyword then
20:         break keyword loop
21:      end if
22:   end for
23: end for
```

In above algorithm 5, We have following variables:

- **TagCounter**: This counts number of tags satisfied for email i.e fraction of its keywords crossed threshold of tag.
- **TagMatched**: This is list which stores tags satisfied for email
- **presentCount**: This counter tracks number of keywords of tag are in dataTree
- **keywordCount**: This counter tracks number of keywords of tag undergo evaluation

We traversing tags and in tags we are traversing every keyword of Tag.keywords list and in it we are checking whether that keyword present in DataTree of trie datastructure, if present we are incrementing presentCount and for each keyword we are incrementing keywordCount, then we are checking conditions as mentioned above in Construction section, i.e if presentCount passed threshold then we will add that tag to TagMatched list and also increment TagCounter, we also check whether TagCounter crossed MAXTAGSALLOWED threshold which is global constant if crossed then return TagMatched list. We also checked whether KeywordCount crossed topKKeyword constant, which represent top K most frequent keywords from keyword list of that tag using frequencyMap attribute.

At line 3, we are traversing all tags lets say that loop run at max t times, then at line 6 we are running keywords loop at max k times and checking whether keyword contains in $O(n)$ where n is length of word. Condition checking and list insertion takes constant time hence TextParser will have time complexity of $O(ntk)$.

Algorithm 6 ModifyTags(KMostFrequent[], tags[])

```
1: for tag in tags do
2:     for word in KMostFrequent do
3:         tag.keywords.add(word)
           // keyword present in email counter
4:         tag.frequencyMap[word]++
5:     end for
6:     tag.keywords.sort(tag.frequencyMap)
7: end for
```

This algorithm adds KMostFrequent words we have obtained from LFU method to tag's keywords list and increment value in frequencyMap with key of word. Atlast we sort keywords list according to frquencyMap's value. Adding of word in list and accessing value in hashmap in constant time. We are doing it for all tags i.e t times and for K times internally but since K is fixed and constant overall insertion of word in list and managing hashmap take $O(t)$. Sorting of list takes $O(log(k))$ where K is number of keywords in list. Hence overall time complexity of ModifyTags is $O(t.log(k))$.

Algorithm 7 LFU(PQ, frequency, word)

```
1: if PQ.size < MAXMOSTFREQUENTWORD then
2:     PQ.add(word, frequency)
3: else
4:     if PQ.top.frequency < frequency then
5:         PQ.pop
6:         PQ.add(word, frequency)
7:     end if
8: end if
9: return PQ
```

In LFU method, we are passing PQ parameter i.e priority queue having min heap internally and managing K most frequent words, we are also passing word and frequency as parameter then we are comparing top of heap with frequency and if it is less than frequency of word we are popping it and then adding word with frequency, both of this operation takes $O(log(K))$ time complexity but since K is constant entire method takes constant time i.e $O(1)$.

Algorithm 8 GetPriority(EmailData)

```
1: TreeNode DataTree = TreeNode();
2: Unordered_map frequency = Unordered_map();
3: KMostFrequent = []
4: PQ = PriorityQueue();
5: for word in Email.Subject.words do
6:     DataTree.add(word)
7:     frequency[word]++
8: end for
9: SubjectPNum[] = TextParser(DataTree)
10: DataTree.clear
11: for word in Email.body.words do
12:     DataTree.add(word)
13:     frequency(word)++
14: end for
15: for word in frequency do
16:     PQ = LFU(PQ, frequency[word], word)
17: end for
18: BodyPNum[] = TextParser(DataTree)
19: while PQ.size > 0 do
20:     KMostFrequency.add(PQ.pop(), word)
21: end while
22: PriorityNums[], TopKTags[] = topKTags(SubjectPNum, BodyPNum)
23: EmailMap.add(Email.ID, PriorityNums)
24: EmailTag.add(Email.ID, TopKTags)
25: ModifyTags(KMostFrequent, TopKTags)
26: return True
```

In GetPriority method, we takes EmailData as parameter that have subject, body of emails with other information. At line 1, we are initializing TreeNode in constant time. At line 2, we are initializing frequency hashmap in constant time that stores frequency of each word in emailData. We also initialize KMostFrequent list and PQ priority queue by default minheap, both to find K most frequent word using LFU method in constant time. Then we traverse words in subject and word in body in line 5 to 9 and line 12 to 16 respectively, where we are adding word to DataTree which takes $O(n)$ where n is length of word, we incrementing frequecy of word in constant time hence for each loop we are taking $O(n)$ time. Let say for subject we are running loop for a times and for body, b times hence both loop will take $O(a.n)$ and $O(b.n)$ time respectively. Once DataTree is prepared at the end of loop we are passing it to TextParser method which return list of at max K tags in $O(ntk)$ time, where t is number of tags and k is number of keywords in it. At line 11, we are clearing DataTree to be used by body loop thats takes constant time. We also calculating most frequent words by iterating over all words in frequency map and passing them to LFU, which takes constant time hence overall time complexity will be $O((a+b).n)$.

Once, tag list is received by calling TextParser method we are storing it in SubjectPNum and BodyPNum list respectively. At lines 18 to 20, we are fetching all words from priority queue to KMostFrequent list by popping nodes that takes $O(K')$ where K' is constant and hence in constant time for K' time and hence overall popping of PQ happens in constant time. Then we will find top K tags among SubjectPNum and BodyPNum by calling topKNums which will use merge technique that get used in mergesort to create list of topKTags and PriorityNums of these topKTags in ordered manner. Merge tachnique of mergesort takes $O(K)$ time where K is max tags allowed i.e length of topKtags list which is constant and hence time is also constant. Then we will add those list to respective lists i.e EmailMap and EmailTag in constant time and will call ModifyTags function that takes $O(t.log(k))$ where t is number of tags and k is number of keywords.

Hence overall time complexity of GetPriority method is $O(max(an, bn, (a+b)n, ntk, tlog(k)))$

i.e $O(ntk)$ which is equivalent to $O(n^3)$. Hence, overall time complexity of algorithm remains unharmed, with multiple optimizations.

4.3 Conclusion

We have seen construction and algorithms above. We have managed to calculate priority in $O(ntk)$ which is equivalent to $O(n^3)$ time complexity i.e unchanged from prior algorithm. Because of multiple optimizations we discussed and shown above, every word gets fair chance to be participate in tag assignment as we always keeping track of K most frequent words from email and adding them to keywords list where also top K' most frequent words will participate in process. And since we are supporting multiple tags, not only one tag but other tags also have fair chance to be part of tag assignment without compromising computational power.

Chapter 5

Future Work

After proper integration of Gmail API for fetching mails and design of the app. We will move onto the following tasks:-

- **Priority Algorithm Integration:** This is one of the main goals of the project. After designing and testing the algorithm, integrating it actually with the app is the next step towards the project completion. This will not only include integration but proper testing along with the assignment of the right color to the respective mails as per estimated priority class.

- **Preprocessor:** Although the final design of the algorithm is a time efficient one but in this world having a faster processing speed is a necessity. What good is an app if it takes 10 mins just to organize your mails. This brings to our next goal after Integration namely Preprocessor. This will preprocess the data(Email body) for us which will in turn help us to reduce the computation time. The novel idea is to reduce the number of characters in email body by removing the punctuation symbols, numbers, the determinants such as 'a', 'an' or 'the' or pronouns such as 'this', 'that', 'these' or 'those' etc. This will further reduce the load of the algorithm all together.

Now we state the further goals which are the addition of features in our application to make

it unique and user friendly. Although the number of features to be added is constricted by time. We present the idea of a few features planning to be added.

- **Text-to-speech:** The idea here to use API to support text to speech, the idea was developed to help users listen to mails. A side of benefit of this feature is to help the blind people.
- **Speech-to-text:** The idea here is again to use API to allow users to write mail using speech. This will increase user convenience. A side of benefit of this feature is to help some physically challenged.
- **Event/ Alarm Setup:** This feature will allow the user to set events via calendar integration or by creating alarms in clock as reminder of event.
- **Google Assistance:** we would like to add google assistance in our app so that the voice related commands are performed smoothly.
- **Date Detection:** The basic idea of this feature was related to the deadline detection. Given a particular context or syntax, the dates may be feasible to be extracted and to give suggestion to users to set up a calendar integrated notification so that they will not forget events or deadlines.

Chapter 6

Results

The progress made in phase II has already been discussed in the above chapters but in order to dive into more depth of the works done, we have included certain images in this chapter. This way it will be easier to judge the output of the work input in the project. So let's jump right into it.

6.1 Sneak Peak at Smail's Interface

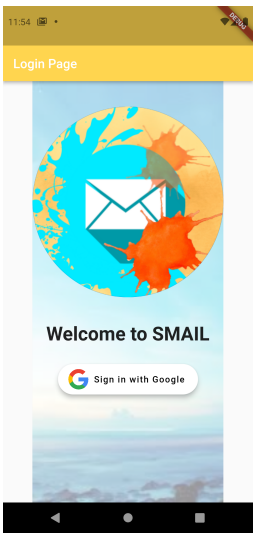

Fig. 6.1 User's Main Page

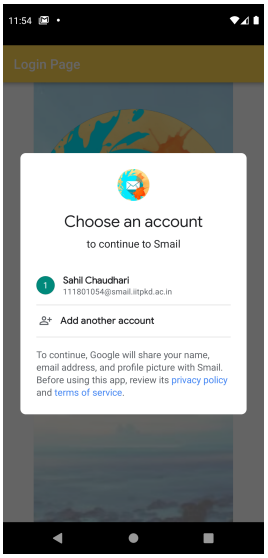

Fig. 6.2 Login Page

Figure 6.1 shows the User's Main Page, The User first is greeted by the main page where our custom designed logo can be seen as welcoming the user with an option to sign in to the respective email id. On the other side, Figure 6.2 shows the Login Page where the user can toggle, choose or add the email account which they wanna use for logging into the app. This interface is somewhat similar to one offered by various google apps, hence is convenient for the user as it might be familiar.

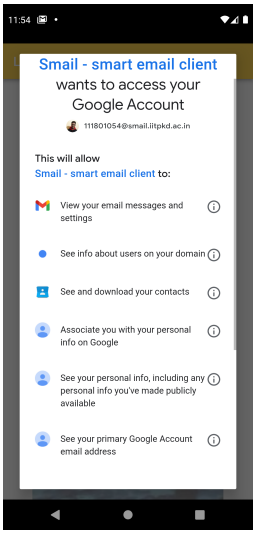

Fig. 6.3 Permission 1

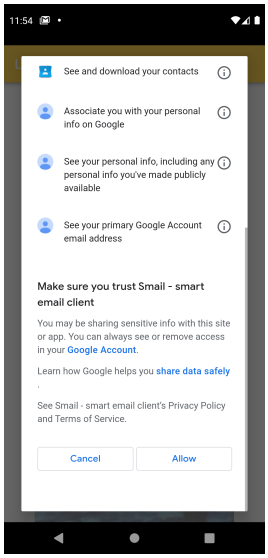

Fig. 6.4 Permission 2

Figure 6.3 and Figure 6.4, shows the permission page, it nothing but the necessary permission that one gives while working with an app. Basically this is what allows us to view mails from gmail or use the other privileges that are associated with google account.

After we go through the permissions, we are all set to fetch the mails, Figure 6.5 shows us the loading page where in the background the app is fetching the mails for the user. Once done we can see the Front Page (in Figure 6.6). It can be seen that the Front page is almost similar to the one shown in Phase I but not entirely, in-fact, it has lot of changes. For instance, each email can now have multiple tags associated with it which can be clearly seen under the emails in the figure. Also the toggle for config page and signout page can be seen. Not only that we can see the color assigned to mails which by default are random. The

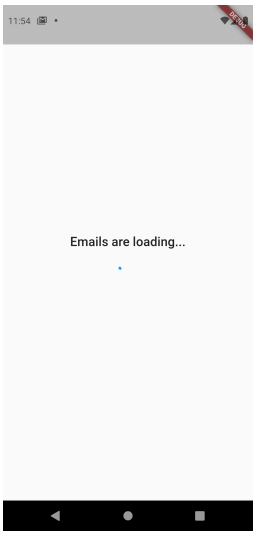

Fig. 6.5 Loading Page

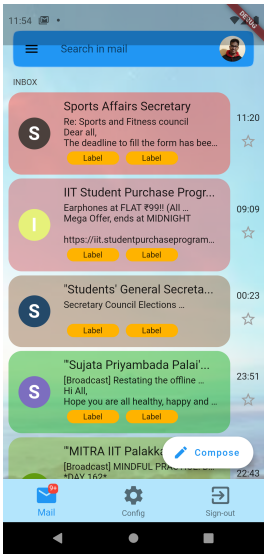

Fig. 6.6 Front Page

cherry on the top is that the interface is now a bit round shaped that the old rectangular one just to increase the aesthetic value.

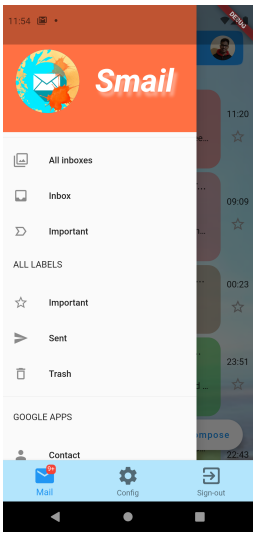

Fig. 6.7 Drawer

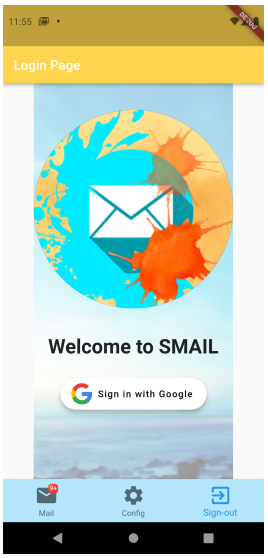

Fig. 6.8 SignOut Page

Figure 6.7 shows the modified drawer, although it is similar to the one which we all have seen in gmail app yet it is somewhat restricted to the use of our application features and also has our designed logo embedded. On the other side, Figure 6.8 shows what will happen once we click on the signout button i.e. we will be redirected to the login page.

Chapter 7

Conclusion

The phase II started with work on Gmail API integration which was a hurdle for the team, since there was less familiarity with that kind of work it took time to overcome this hurdle. But as they say no task is an impossible task, we just needed more effort from the team and as we see in the end, we were able to achieve our goal. In Phase II, we studied about Gmail API, and integrated it with our application as a result of which we were able to fetch emails and display them in our application. In this phase, we also finalised the priority algorithm and modified it to the extent where we find it acceptable to integrate with the application. Also, we made substantial changes in the front-end of the application to make it more unique so that we can proudly say that it belongs to us.

But this is not the end, we still have miles to go. We are clear about our future goals:- we plan to integrate the priority algorithm in the application and create a preprocessor to make the processing of priority handler faster by reducing the data that needs to be processed. And then we will start adding feature in our application as discussed in above chapters.In the end, It is safe to say that the project has taken the optimal pace in the direction of the right path.

7.1 Contribution

The project has been a shared effort of both the teammates, helping each other and facing the hurdles together. Although if we get down the specifics of the contribution in phase II, this can be said. The modification to the priority algorithm which includes the brain storming of ideas and implementation was done together by us both. Apart from that Sahil worked on the gmail API integration whereas Vishesh worked on the front end. Both of us helped each other to overcome each stepping stone and promise to do it until the project is over.

References

[1] "Gmailapi gmail.v1 library documentation." [Online]. Available: https://pub.dev/documentation/googleapis/latest/gmail.v1/gmail.v1-library.html

[2] "Mime protocol's multipart content type documentation." [Online]. Available: https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html

[3] "Googleapis authorization scopes information." [Online]. Available: https://developers.google.com/identity/protocols/oauth2/scopes

[4] "Flutter widget documentation." [Online]. Available: https://docs.flutter.dev/development/ui/widgets-intro

[5] "Least frequently used replacement policy." [Online]. Available: https://www.geeksforgeeks.org/least-frequently-used-lfu-cache-implementation/