

APPROVAL SHEET

Title of Thesis: Enhancements for the Search Functionality of an Open Source Email Client

Name of Candidate: Aishwarya S Bhide
Master of Science, 2016

Thesis and Abstract Approved: _____
Dr. Charles Nicholas
Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

ABSTRACT

Title of Thesis: Enhancements for the Search Functionality of an Open Source Email Client

Aishwarya S Bhide, Master of Science, 2016

Thesis directed by: Dr. Charles Nicholas, Professor
Department of Computer Science and
Electrical Engineering

Email is one of the most popular communication media in today's world. Various email clients are extensively used for professional as well as personal use. Many times the users of these email clients have to search for emails they have received or sent in the past. Although all email clients support search functionality, most of the widely used email clients do not have many advanced features. These features include query auto-correction, and showing emails related to an open email, which make email search more user-friendly and effective. The goal of this thesis is to identify such limitations for one such email client, i.e. Thunderbird, and to create an add-on to address these limitations.

Enhancements for the Search Functionality of an Open Source Email Client

by
Aishwarya S Bhide

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2016

I dedicate my work to Aai, Baba, Tai and Mandar

ACKNOWLEDGEMENTS

I would like to first express my deepest gratitude to my thesis advisor, Dr. Charles Nicholas for supporting me through my masters study and research. I am gratefully indebted to his invaluable guidance, understanding, patience and motivation for this thesis. I would also like to thank my parents Subhashchandra Bhide and Prasanna Bhide and my sister, Harshada Bhide for being my strength. Last but not the least, a big thank you to my best friend and fiance, Mandar Haldekar for his unfailing support and continuous encouragement. This accomplishment would not have been possible without them. Thank you!

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
LIST OF FIGURES	vii
Chapter 1 INTRODUCTION	1
1.1 Email clients and their search capabilities	2
Chapter 2 RELATED WORK	7
Chapter 3 QUERY AUTO-CORRECTION	11
3.1 Experiments performed	13
3.1.1 Word level spelling correction	13
3.1.2 Query level spelling correction	16
3.2 Thunderbird add-on	21
Chapter 4 RETRIEVING RELATED EMAILS	31
Chapter 5 CONCLUSION AND FUTURE WORK	35
5.1 Conclusion	35

5.2	Future Work	35
	REFERENCES	37

LIST OF FIGURES

3.1	Architectural diagram of add-on	22
3.2	Snapshot of input "thesis" to Thunderbird global search	23
3.3	Snapshot of output of Thunderbird global search for query "thesis"	24
3.4	Snapshot of input "thesis" to add-on	25
3.5	Snapshot of output of add-on for query "thesis"	26
3.6	Snapshot of output of Thunderbird global search for query "social media" . .	26
3.7	Snapshot of output of add-on for query "social media"	27
3.8	Snapshot of output of Thunderbird global search for query "social media" .	27
3.9	Snapshot of output of add-on for query "social media"	28
3.10	Snapshot of output of add-on for query "travel details"	28
3.11	Snapshot of output of add-on for query "travel details"	29
3.12	Snapshot of output of add-on for query "university orlando"	29
3.13	Snapshot of output of add-on for query "university orlando"	30
4.1	Input email for related emails	32
4.2	Output without sender name in search query	33
4.3	Output with sender name in search query	33

4.4	Input email for related emails	34
4.5	Related emails for email in Fig. 4.4	34

Chapter 1

INTRODUCTION

Email systems have become a part of day-to-day life not only for the people working in science and technology related fields, but also for people working outside those fields. Emails are used to communicate professional as well as personal messages. They are also a very commonly used means for sharing information and data in various forms e.g. photos, videos, hyperlinks, documents, etc.

Many people also use emails as storage for this information. This facilitates the user to access this stored information regardless of time and location, the only requirement being availability of an internet connection to access the emails. Emails are particularly useful as storage as the information being stored gets stored on the server, so the user's device memory does not get used up, which is a great advantage for a user accessing emails from small electronic devices like their smart phones. Since the users tend to have their smart phones with an internet connection available most of the time, they have continuous access to the information stored on email server. Also, since most modern email systems provide large amounts of server memory for each user without cost, emails are an attractive solution for storing information like copies of documents like passports, driving license, photos etc.

A very important feature for use of emails as a communication medium or as a means of storing and sharing of information is the search functionality in email systems. The users

frequently need to search and retrieve past emails received or sent by them or information stored in emails by them. As the email data in user accounts is composed of a large number of emails, scanning and filtering it for search is computationally very expensive. Most email systems in use currently provide text based search functionality, though it is not satisfactorily advanced and user-friendly.

This thesis tries to find the limitations of search functionality and implement the solution for one such email client, namely Thunderbird. The limitations addressed in this thesis are auto-correction of the search query and finding emails related to currently open email.

Misspelling search queries is very common in text based search systems. In Thunderbird, if the user misspells a query, no search results are shown. If the misspelling is auto-corrected, user experience with the search functionality will be greatly improved.

Also, when a user opens an email, he or she may want to look for another email related to it. If emails related to currently open email are shown in a tab beside the open email on clicking a button instead of having to search for them, the user can easily access the required emails without having to type a search query for it. The related email that user is interested in can be opened and seen simultaneously with the currently open email making reading the emails more efficient.

1.1 Email clients and their search capabilities

Email clients are programs used to access and manage user's emails. There are many commercial as well as open source email clients available. Some popular examples are: Microsoft Outlook, Opera Mail, eM Client, and Mozilla Thunderbird. There are also some command line email clients without GUI for users who prefer to manage their emails without switching from the terminal to a GUI window e.g. Mutt, Alpine, and Sup. All of these email clients provide text search functionality which lets the users retrieve their emails

and data using keywords related to them. However, the search functionality of these email clients can be improved by adding some features which make email access more effective.

Though some programmers may prefer command line email clients, most users are not much familiar with terminals and command line. GUI based email clients have a much larger user base as they are more intuitive and easy to use. So this thesis considers one such GUI based email client, Thunderbird.

Thunderbird is an extensively used, free and open source email client developed by Mozilla. The source code of Thunderbird is in CPP, though extensions (TBe) providing additional functionality can be written using Javascript and XUL (XUL). XUL stands for XML User Interface Language, which is Mozilla's XML-based language for building user interfaces of applications.

Thunderbird is a good option when developing new features for an email system as it is open source and its documentation is available on internet which provides guidance on creating add-ons for Thunderbird. It is particularly useful in this case as it provides the feature of text search on emails using different search engines. It also includes a message indexing and search system called "Gloda" which provides full-text search capabilities and displays faceted search results.

Users get a lot of emails every day. So it becomes hard to organize and keep track of the emails. To help users to search emails effectively, Thunderbird provides many configuration options. The users can enable the operating systems search to search emails. They can enable or disable the Global Search feature. Also for IMAP folders, users can decide whether messages in that folder will be included in global search or not.

When a user types a query having multiple words, Thunderbird shows the results which contain at least one occurrence of each of the word in the query. If user encloses multiple words in quotation marks, Thunderbird returns search results containing messages having all the words in the order they are specified in the search field i.e. it searches for the

whole phrase in quotation marks instead of searching for each word in the phrase. These two types of searches can be combined. For example, if user types the term "computing" and the phrase "IEEE conference", Thunderbird will find the messages that contain both the term and the phrase.

Apart from the search configuration options, following are some features which Thunderbird provides for storing and looking up emails:

- Quick search: Thunderbird provides quick search bar where user can look up an email from current folder by using the subject or sender's name.
- Smart folders: These are saved searches which are displayed as folders. The search can be based on multiple criteria.
- Tags: Tags let users categorize their emails or give them priorities. Some predefined tags in Thunderbird are Important, Work, Personal, To Do, Later. User can look up emails using tags.
- Filters: This feature is useful for performing specific actions on emails matching certain criteria. These actions include tagging emails, marking emails, forwarding emails, deleting emails, storing emails in a specific folder etc. This can be used for organizing emails for easier lookup.
- Views: Thunderbird has 5 different views: All, Unread, Favorites (defined by user), Recent (recently viewed folders) and Unified (multiple accounts).

Apart from these features, Thunderbird also has a search functionality called open search, which lets the users search a term on web using different web search engines. Also, there are many add-ons created by Thunderbird community which provide many more features.

Even though it has all these features, Thunderbird lacks some very basic functionality which is important for efficient lookup of emails. Some extensions / add-ons are available to improve search experience with Thunderbird. Following are some of such add-ons:

- Expression Search / GMailUI:

This add-on is based on the search functionality provided by Gmail. It allows the user to search for their emails using expressions. This is a powerful way to search emails for users who receive a large amount of emails. The users can search emails using expressions like "from:", "to:", "subject:", "attachment:" etc. Users can also search a combination of these search expressions. Also, it provides the feature to search similar messages on clicking a subject/from/recipient.

- Unified Search:

This add-on unifies the global search functionality and quick filter feature of Thunderbird.

- Remove Duplicate Messages:

This add-on helps to identify and remove duplicate messages in folders and subfolders.

Even though there are many such add-ons available for Thunderbird, Thunderbird still lacks some features. e.g. it does not show emails related to currently open email based on the content of email. The GMailUI extension has a click to search functionality which lets users search for selected subject or address of sender or recipient. Though, searching using sender/recipient may return a large number of results out of which only few may be related to open email. Also, searching for subject may not always return related results as many times subject of email does not represent the content of email very well. Moreover, having

to select the subject/Sender/Recipient and then press Ctrl/Shift + Right Click to search is not as user friendly as clicking a button and getting results related to content of open email.

Also, a feature which autocorrects user's text queries will improve the user experience significantly. Thunderbird does not use synonym search while searching for emails, which would provide better results. If the search functionality searched attachments of emails for content related to query, it will be more effective.

Out of the improvements listed above, this thesis tries to implement the feature of auto-correction of search queries as an add-on for Thunderbird to improve the email retrieval, as it has a large impact on user experience. It also implements the feature of retrieving and displaying related emails based on content of open email on click of a button.

Remainder of the thesis is organized as follows: Chapter 2 provides a summary of related work done in auto-correction of queries and finding related emails. Chapter 3 describes experiments performed for query auto-correction and the method used to provide it as a feature for Thunderbird. Chapter 4 describes the method used for retrieving emails related to open email. Finally, the thesis concludes with future work and a summary of the experiments performed.

Chapter 2

RELATED WORK

Many times users misspell the text queries when searching. The misspelling of words involves missing characters, inserting extra characters, or changing a character to some other character. Also, queries having multiple words may have errors like two words merged together i.e. missing a space or splitting a word into two words i.e. extra space inserted between a word. In such cases, the system would provide no search results or irrelevant search results.

Hence, automatic spelling correction for queries is an important feature in search systems which can improve user experience, save time and increase productivity. Most leading web search engines provide the functionality of spelling correction for queries based on the query logs of a large number of users. This approach works well for web search as it leverages the collective intelligence of all these users and the collection of documents to be searched i.e. web is common and available for all users.

Cucerzan and Brill (Silviu Cucerzan 2004) used the above approach to iteratively transform input query strings into other strings which are more likely to be correct queries according to statistical information of internet search query logs. They observed that roughly 10%-15% of queries sent to web search engines have errors. Most web queries are not well formed sentences and may contain words that are legitimate but not present

in a dictionary e.g. nouns like Shrek, Nemo etc. Hence, the validity of a query cannot be determined simply by looking up the words in query in a dictionary or by checking if it is grammatically correct. So they proposed to use query logs of all users to check validity of a word from its frequency in what people are searching for. The challenge with this approach is that many frequently occurring queries in logs are misspelled.

Google search engine figures out possible misspellings and their likely correct spellings by using words it finds while searching the web and processing user queries. So, unlike many spelling correctors, Google can suggest common spellings for words which are not in a dictionary like proper nouns of people, organizations and places, slang words, acronyms etc. The search in Gmail also uses the same auto query correction functionality as Google search. So when user misspells a word, it corrects the word based on web search data and not based on user's emails. e.g. if user types 'urgwy' to search for an email containing the word 'uruguay', Gmail search may not find the email and instead ask whether user meant 'rugby' or 'uruguay'.

Thus, this approach of query correction based on query logs is not very useful in email search queries as the query log for any one user is usually not large enough and the query log for other users is not useful when trying to check correctness of the query entered by a particular user as emails to be searched and search intents are highly specific to the user. Also, the emails of a user are private and available only to them. Hence, the query corrections need to be personalized and context specific i.e. based on user's current email state.

Taking this into account, Bhole et al (Abhijeet Bhole 2015) have considered the problem of providing spelling corrections using user's own email data. They have developed a system called SpEQ which uses machine learning to find spelling corrections directly from the user's own email data. It works equally well on different sizes of email data and works without having to know user's search history. Their system employs a ranking framework

based on query and its context. It generates candidates for misspelled queries using dynamic programming and a scoring function based on inverse document frequency (IDF) and edit distance of the candidate's words. The set of feature functions that is used to score the candidates is based on lexical similarity with original query, query context, content of mailbox and search context. Content based features are based on the statistics of candidate tokens in subject lines and contacts, number of emails fetched by a query (if a query fetches too many emails, it is unlikely to be useful). Contextual feature functions represent current state of mailbox i.e. they are calculated over recently accessed emails.

Bao et al (Zhuowei Bao 2011) have proposed a graph approach to spelling correction in domain centric search where the domain could be anything like desktop, email or large scale web sites. Their system generates candidates for each word based on edit distance. All candidates of a word are given scores calculated using weighted Damerau-Lavenshtein edit distance, phonetic similarity, the candidate's existence in English words, logarithm of the candidate's frequency in corpus etc. The weights for these features are found using Support Vector Machine (SVM). Their algorithm called MaxPaths returns a set of candidate suggestions for a query based on the score. For this, it finds strongly plausible tokens for each word, constructs the correction graph which has edges from every candidate of each word to every candidate of its next word in query, finds the top k paths with highest weights in the graph and then re-ranks the paths using word correlation.

Many times when users open and read an email, they may need to refer to some previous email about the same topic, most probably sent by the same person as the current email. e.g. User may have received information about a job position and applied for the position, they may receive an interview call a few weeks later and may want to check the email describing the position when reading the email regarding interview. In such cases, it will be useful to have related emails searched on the click of a button instead of typing the search query.

Finding related emails based on email content is similar to finding related documents from a corpus. Though the purpose of most algorithms which try to find similar documents is to remove duplicates. In this case, we want to find emails which are related to open email, not duplicate emails. In most cases, the related emails will not have the exact same text chunks as open email. In fact, the related emails we are looking for are presumed to contain more information about the topic of open email. So the similarity detection algorithms used for finding duplicate documents will not work for our purpose. Instead, we have to try to find keywords from the email content and use them to search for related emails.

Chapter 3 describes the experiments performed for auto-correction of queries based on current state of email account and how this feature works in our Thunderbird add-on.

Chapter 3

QUERY AUTO-CORRECTION

Thunderbird text search does not have a query correction mechanism. So if the user misspells a query, the search may give no search results or irrelevant results. Also, the multiple word text queries entered by user show only those emails in results which contain at least one occurrence of each of the word. So if the user misspells even one word of the query, the search will return no results even if the query contains other correct words. The query auto-correction feature implemented as add-on in this thesis helps get better results by correcting the misspelled words. Even when a misspelled word cannot be corrected by the add-on, it shows the results based on the correct words.

For implementing this feature, various experiments were performed on the Enron-Random dataset (enr) which contains randomly sampled emails from Enron corpus.

The experiments which required machine learning were performed using Scikit Learn (Buitinck *et al.* 2013) which is a Python package which provides various machine learning algorithms. The experiments performed used the machine learning algorithms SVM and Logistic Regression.

For training the Machine Learning algorithms, Peter Norvig's list of words and their common misspellings was used which is available online (nor). This list contains approximately 7800 words and their possible misspellings. The training set included approxi-

mately 6000 words and the test set included approximately 1800 words. Out of this list, only the entries where the correct word is present in the index generated for searching the email corpus were taken into consideration. The lines in the training and test files were in the format:

CorrectWord: comma separated list of misspellings for the word

e.g. dictionary: dictionery, ductioneery, dictionary

The indexes for the corpus were generated using Whoosh (who) which is a full text indexing, search and spell checking library in Python. Whoosh was also used for suggesting candidates for word correction based on edit distance and document frequency of the candidates. The document frequency of a word and the documents containing a word were found using Whoosh functions. For implementing the correction of queries, three separate indexes were generated, one index each for the content of emails, the subject of emails and the contacts in the 'to' and 'from' fields of the emails. As the Enron emails are in text document format, the subject and to-address, from-address of each email was found from the header by parsing the metadata sections of emails.

For checking if a string was a word existing in English language, Natural Language Toolkit i.e. NLTK (NLT) was used. NLTK is a Python package for natural language processing tasks.

For evaluating the query auto-correction feature, a list of queries had to be extracted from the Enron-Random emails. Multiple queries were formed for each email by taking random strings in the emails and introducing randomized errors in them. For finding the keywords in each of these strings, Topia.termextract (top) was used which is a Python package which implements content term extraction. Given some text, it finds important terms in that text using Part-of-Speech tagging and frequency of terms.

For assigning weights to the correct word candidates returned by Whoosh for each misspelled word, following features were considered:

- **Levenshtein Distance:** The Levenshtein distance is a metric for measuring the amount of difference between two sequences i.e. an edit distance. The Levenshtein distance between two strings is the minimum number of edits needed to transform one string into the other, where the edit operations possible are: insertion, deletion, or substitution of a single character. For example, the Levenshtein distance between "kitten" and "kitchen" is 2, since first word can be changed into second word by substituting second 't' with a 'c' and inserting an 'h' after the c. There is no way to do this transformation with fewer than two edits. The Python package Levenshtein (lev) was used for finding Levenshtein distance between two words.
- **Double Metaphone:** Metaphone is a phonetic algorithm for indexing words by their English pronunciation. In this algorithm, similar sounding words share the same keys. It uses information about variations and inconsistencies in English spelling and pronunciation to produce encoding for words, which can be used to match names and words that sound similar. Double metaphone is based on the Metaphone algorithm. While Metaphone is applicable only to English, Double Metaphone takes into account more languages. The python package Metaphone (met) was used for finding Metaphone for the words.

3.1 Experiments performed

3.1.1 Word level spelling correction

Experiment 1: Following steps were performed on the query containing words w_1 to w_n :

For each word w_i , multiple candidates were generated by using edit distance and document frequency of generated candidates. This was done using spelling corrector function-

ality of Whoosh. The Whoosh function takes the original word w_i , the index and maximum allowed edit distance as input. It returns a list of words having edit distance less than maximum edit distance specified from w_i and which are present in the index. The candidates are generated from each of the three indexes (content index, subject index and contacts index). The original word w_i is also considered as a candidate.

Weights are assigned to each of the candidates based on the features:

- Existence of the candidate in content index
- Existence of the candidate in subject index
- Existence of the candidate in contacts index
- Whether the metaphone encoding of candidate matches with that of original word
- The Levenshtein distance of candidate from original word
- Document frequency of candidate

We used document frequency as feature instead of term frequency because a candidate present in multiple emails is more likely to be what user is searching for as opposed to a candidate present multiple times in a single email. Each of the above features was assigned a manual weight empirically based on importance of the feature after different combinations of feature weights were tried out. Various words and their misspellings were used for trying to determine weights of features and for testing. As each feature's importance differed based on the errors in spellings of input words, the weights of features which worked well for a query did not necessarily work well for another query.

The candidate having the maximum weight is returned as the most probable correct word for each word. The most probable candidates for each word replace the original word in the query and the query thus formed is returned as corrected query.

After trying out different weights for different features, the maximum accuracy achieved was approximately 70% for test data consisting of randomly generated words and their misspellings from the email corpus. The random misspellings were generated by inserting a random character, deleting a random character, changing a random character to another random character or swapping two random adjacent characters in the word. The merge and split errors were not considered for word level query correction.

Experiment 2: Based on the above observation, weight of each feature differed for each misspelling with respect to correct word. So to decide the correct word for each word, Machine Learning models were trained by taking into account same features as in Experiment 1 above. The models tried to predict whether a candidate could be the correct word corresponding to original word. This approach was based on word level correction part of MaxPaths algorithm of (Zhuowei Bao 2011).

The features corresponding to existence of candidate in the three indexes had binary values i.e. value for the feature would be 1 if the feature existed in that index, 0 otherwise. Similarly, if the metaphone encoding of a candidate matched that of the original word, the value of metaphone feature for that candidate would be 1, otherwise it would be 0. The Levenshtein distance feature had the value same as that of Levenshtein distance between original word and candidate. Similarly, document frequency feature has value same as document frequency of the candidate.

The models tried to classify words in 2 classes. Class 1 is class of correct spellings for original word and class 0 is for incorrect candidates of original word. The features for each misspelled word and its candidates were found and the candidate matching the correct word for the misspelling was classified as belonging to class 1 and other words were classified as belonging to class 0. The data used for training and testing was taken from Peter Norvig's list of words and their common misspellings.

The total number of words including both correct and incorrect words in test set were 58664 out of which only 1307 were correct spellings. Two machine learning algorithms i.e. SVM with RBF kernel and Logistic Regression gave accuracy of approximately 98% for predicting a word to be correct or incorrect. Though the accuracy was good, SVM predicted only 629 words to be correct while 1307 correct words were present in the test set. Similarly, Logistic Regression predicted only 721 words to be correct out of the 1307 correct words in test set. So the recall is approximately only 55%. The reason for less recall was that the dataset was unbalanced.

Experiment 3: This experiment followed the same procedure as experiment 2 above, except that only those candidates of a word were considered which were within an edit distance of \log_2 (length of word). This was done to test if precision and recall increased by considering only the words which were more likely to be correct words, since 80% of spelling errors are within edit distance 1 and almost all spelling errors are within edit distance 2 from the correct word. Though, instead of keeping maximum edit distance as 2, it was changed based on length of the word as longer words may have more spelling errors.

The accuracy remained almost same as experiment 2, while the recall decreased as SVM classified 686 words as correct and Logistic Regression classified 692 words as correct from the 1307 correct words.

3.1.2 Query level spelling correction

In the experiments for query level spelling correction, the errors caused by splitting and merging of words were also considered in addition to the errors mentioned at word level correction.

For this application, the execution time for correcting each query is as important as the

accuracy of correcting queries. If a query takes a long time to be processed for correction, user may find out the error in query, correct it and search for the corrected query in shorter amount of time. Also, when the original query entered by user is correct, the application may take a long time to determine that it is correct or it may determine the query is incorrect and return a corrected version which is different from what the user wants to search. Considering these factors, the query level spelling correction implemented is different than MaxPaths, since MaxPaths would be computationally very expensive and would take a lot of time to process each query.

Experiment 1: In this experiment, each query from test set was parsed and the words which are English words or which are in the index are kept as they are because they are likely to be correct and trying to find correction for them is more likely to return words that are different from what user wanted to search. Also, this saves significant amount of processing time for each query.

The candidates for remaining words are generated using the experiment 3 in word level spelling correction section above. Here, experiment 3 was used instead of experiment 2 above as most misspelled words are within edit distance 2 from correct word. The machine learning model learned for word level correction is used to find if any of the candidates is predicted as correct for each word. If we find a correct candidate, the word is replaced by that candidate. This is useful for correcting the word misspellings that do not include merged or split words.

For the words which do not have any candidate predicted as correct, we need to check for merge-split errors. For this, all such words which are consecutive are considered as one term obtained by concatenating these words in original order without spaces separating them. For every such term, we find all possible combinations of words that can be formed by inserting spaces in the term.

e.g. If a term is "abcd", we get all possible combinations as: "abcd", "a bcd", "ab cd", "abc d", "ab c d ", "a b cd", "a b c d" etc.

Every such combination is tried as a query_option. For each query_option, a weight is calculated based on following features:

- existence of each word in English or in index
- edit distance of query option from original term
- difference between number of words in original term and query option
- number of documents containing all words in query_option

If a word is not existing in English or in index, it's candidate having maximum probability to be correct according to above machine learning model replaces it in query option. The query_option with maximum weight replaces all the words in term when forming corrected query.

For testing this system, random strings were taken from each email as queries after removing the email headers containing metadata of the email. Random errors were introduced at random positions in each of the queries. Each query contained single or multiple errors. The errors consisted of inserting a character at a random position, deleting a random character, swapping two adjacent characters, changing a character to another character, inserting a space at a random position, deleting a space from the query.

After testing this system using the test data created, accuracy of this approach was approximately only 18%. It was observed that the accuracy was very less as random lines from emails do not represent search queries properly. Most of the random lines consisted of many stopwords, special characters and no keywords. On the other hand, users tend to type keywords as search queries when using text search. In this approach, queries like "I

will see you on Monday”, ”thank you” or ”/da” were formed which are not representative of actual search queries.

Experiment 2: Based on the above observation, instead of taking random lines as queries, keywords were found from such random lines and were used as queries. Keywords were found based on Part-of-Speech tags and term frequency using the Topia library. The same procedure as above was used to correct queries. The accuracy did not improve much over the previous experiment, but the quality of queries in test set was much better.

Both experiments mentioned above consisted of finding all possible combinations of terms by inserting spaces in the terms which requires a lot of processing time for each term. Also, finding candidates for all words in query or in any combination of term, which do not exist in index or in English language is another computationally heavy part, as it requires finding all words which are within given edit distance from original word. Finally, training machine learning model based on thousands of words and predicting correctness of a word based on that model takes a lot of time and memory for processing. As mentioned above, keeping the processing time short is important for this application.

Also, following are some common observations from the queries:

- Most misspellings formed by merging words together consist of two words merged together. It is very uncommon for more than two words to be merged together as spelling error.
- In most split errors, one extra space is inserted between a word. It is very uncommon for a word to be split in more than 2 words as spelling error.
- If a word is formed by a merge error, the two words being merged are less likely to have any other spelling errors.

- A word can be misspelled in a number of ways. Also, for any misspelled word, multiple candidates may be words existing in English or in index. In such cases, the candidate having lesser edit distance from word may not always be the correct one. Similarly, the candidate having same Metaphone encoding as the word may not be the correct one. Thus, it is very hard to learn machine learning models that achieve high accuracy from these features, as for each misspelling, different features have different weights. There does not exist any pattern in misspellings. We can only try to predict the correct word from common observations.

Experiment 3: Based on the observations above, the machine learning models were eliminated. The features for word level correction were given weights manually based on common observations. Also, instead of finding all possible combinations of a merge-split term which could contain any number of words, only those combinations containing two words were considered. In such combinations, words existing in English or in index were given more weight as user is less likely to make another error in two words having merge-split errors. The most probable candidates for words not in English or in index were given less weight in comparison.

After these simplifications, the processing time for each query was significantly shorter and 584 out of 680 random queries taken from the Enron-random corpus were corrected properly with accuracy of approximately 86%.

Bao et al (Zhuowei Bao 2011) used a Java API to find Google's accuracy for correcting queries in their datasets. The accuracy was approximately 50% for email dataset, while it was approximately 80% for site search. Our experiment gave a much better accuracy of approximately 86% for email dataset. Also, Thunderbird can be used for multiple email systems i.e. we can use it to access various personal as well as corporate email accounts, while Gmail's auto query correction is useful only for Gmail accounts. Hence, our add-on

provides better user experience.

3.2 Thunderbird add-on

Thunderbird add-ons are written using javascript and XUL (XML User interface Language). The add-on created for this study uses the system in experiment 3 for query level correction. Though, the system requires many modifications for its integration with Thunderbird. Following are some of the differences between the system of experiment and the add-on:

- As the Enron-Random dataset has emails stored as a text file per email, Whoosh was used to build an index and other processing. Thunderbird downloads and stores the emails in user account only if POP protocol is used. In case of IMAP protocol, which is widely used, Thunderbird does not download the emails. Instead, they are accessed from the server. For using Whoosh in case of IMAP, all emails would have to be downloaded which defeats the purpose of using IMAP for email access.

So the query auto-correction add-on uses the Gloda (short for Global Database) indexing system provided by Thunderbird. Gloda uses a SQLite database to store all the information regarding emails. The system uses SQLite queries to find document frequencies of words and documents containing words.

- SQLite has a FTS4 extension called FTS4aux which supports full text search. This is used to create a table with all terms in all emails.
- Spellfix1 virtual table is used to search words closely matching a word from all the terms in emails. Spellfix1 is a loadable extension. The correct word candidates are returned by Spellfix1 instead of using Whoosh suggest function.

- As the add-on is implemented in javascript, it needs to call the Python program for correcting entered query. nsIProcess is used for this purpose.
- GlodaMsgSearcher is used for calling the global search of Thunderbird on corrected query.

Figure 3.1 shows the architectural diagram explaining the various components of add-on. The UI components are defined in XUL files. When we add this add-on to Thunderbird,

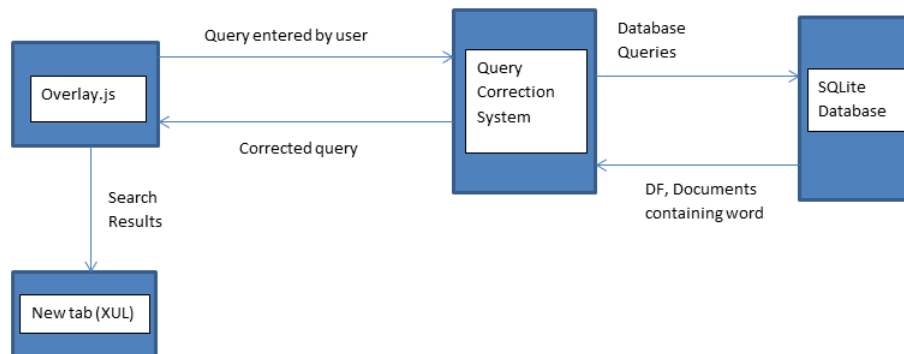


FIG. 3.1. Architectural diagram of add-on

it inserts the search box in the lower right corner of main window of Thunderbird. When the user enters a query in the box and clicks submit button, the function in overlay.js is

called. This function uses nsIProcess to call the Python program which executes the query correction. The Python program interacts with the Gloda database to find the document frequencies of words, candidates for misspelled words and IDs of documents containing words. All query candidates are weighted using same features as experiment 3 above. It returns the query with maximum weight as correct query. As nsIProcess does not have a mechanism to get output from the called program, the Python program writes the corrected query in a file. Overlay.js reads the corrected query from the file and calls the global search using GlodaMsgSearcher which shows the search results in a new tab.

Following are some examples of the outputs given by the extension developed and Thunderbirds global search for same inputs for word level correction:

- For a single word misspelled query "thesis" entered in Thunderbird's global search box, we get no results as shown in figures 3.2 and 3.3.

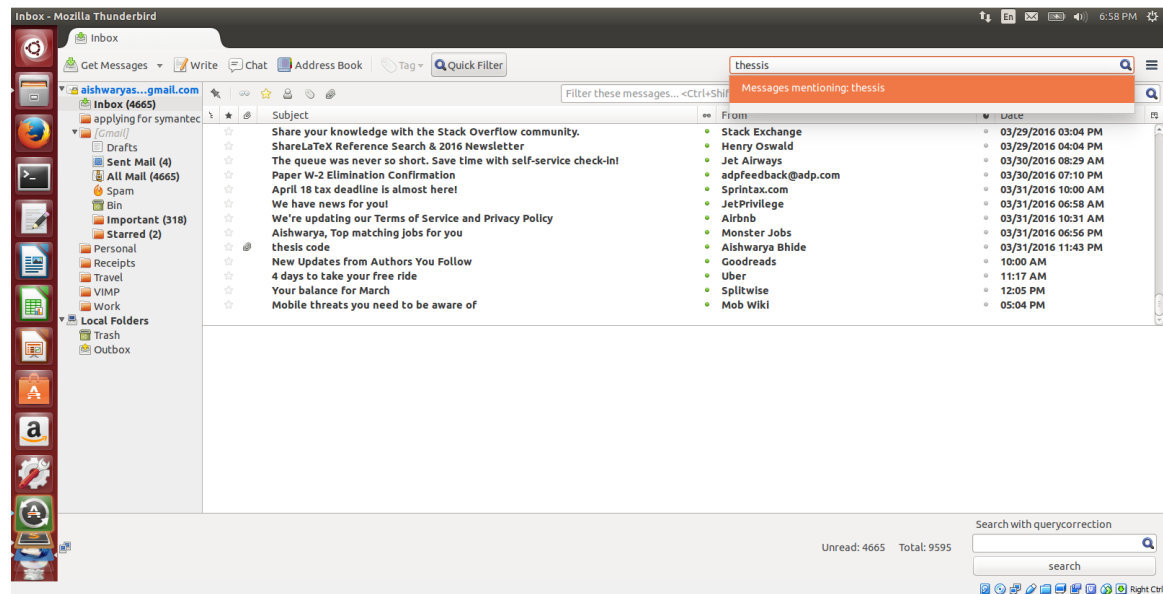


FIG. 3.2. Snapshot of input "thesis" to Thunderbird global search

While, the same query is corrected and user gets correct results when entered in the

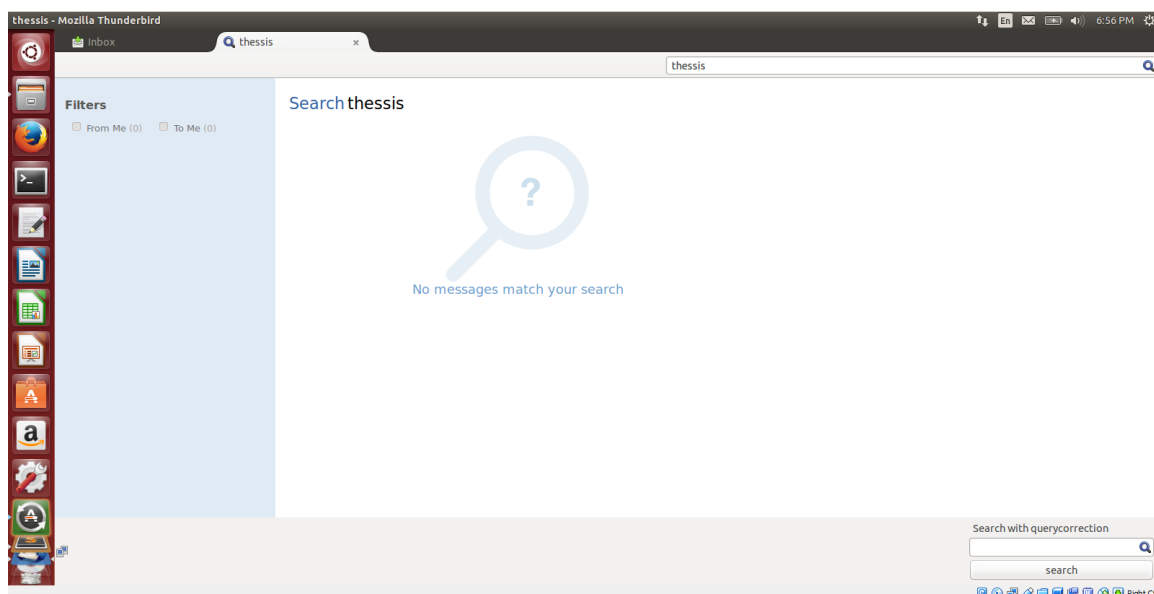


FIG. 3.3. Snapshot of output of Thunderbird global search for query "thesis"

querycorrection add-on search box as shown in figures 3.4 and 3.5.

- For the query "socil media" involving multiple words out of which one is misspelled, Thunderbird gives no results as shown in figure 3.6.

While, the add-on gives correct results by correcting the misspelled word as shown in figure 3.7.

- For the query "socil medea" where both words are misspelled, Thunderbird does not give results as shown in figure 3.8.

While, the add-on corrects both words and gives results as shown in figure 3.9.

Similarly, following are some examples of the outputs given by the extension developed and Thunderbird's global search for same inputs for query level correction:

1. For a query where two words are merged like "traveldetails", Thunderbird global search fails to return results as shown in figure 3.10.

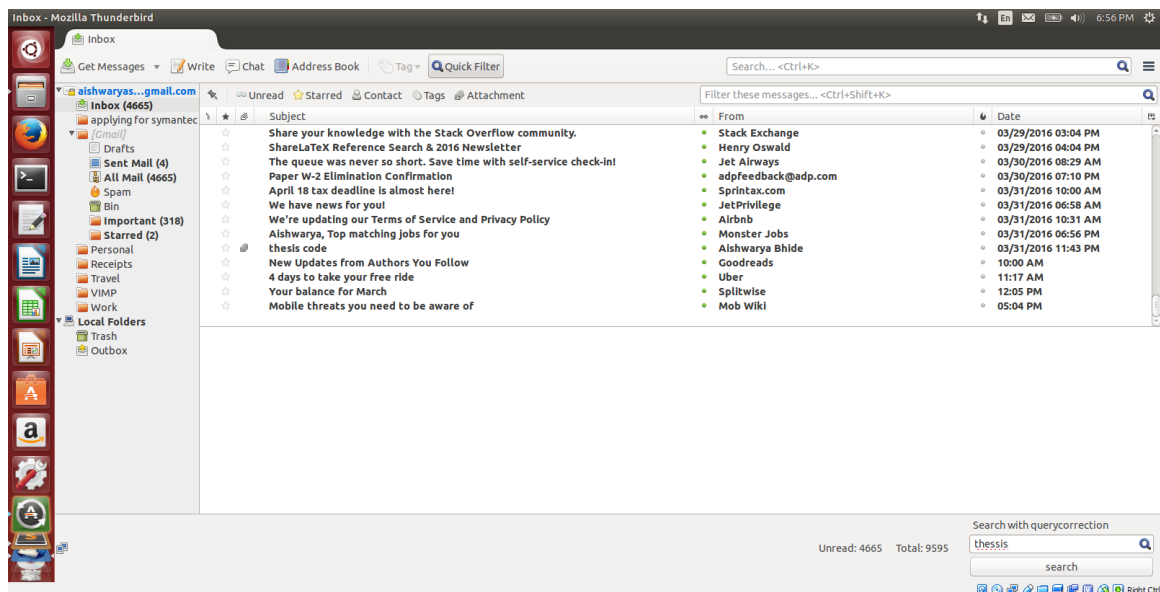


FIG. 3.4. Snapshot of input "thesis" to add-on

Whereas, the add-on split the word and returned results as shown in figure 3.11.

2. For the query "unive rsal orlando", where the word "universal" is split into two words, Thunderbird fails to retrieve results as shown in figure 3.12.

Whereas, the add-on attempts to correct the spelling giving output query as "Orlando universe l" and returns results as shown in figure 3.13.

Here, although the add-on did not give the correct query "universal orlando" as input to the search, it changed the query to "Orlando universe l" which is close to the correct form of original query. So user would get relevant results instead of empty results. Thus, the add-on gives better results than Thunderbird global search in case of misspelled queries. If a query is spelled correctly, it keeps it as it is.

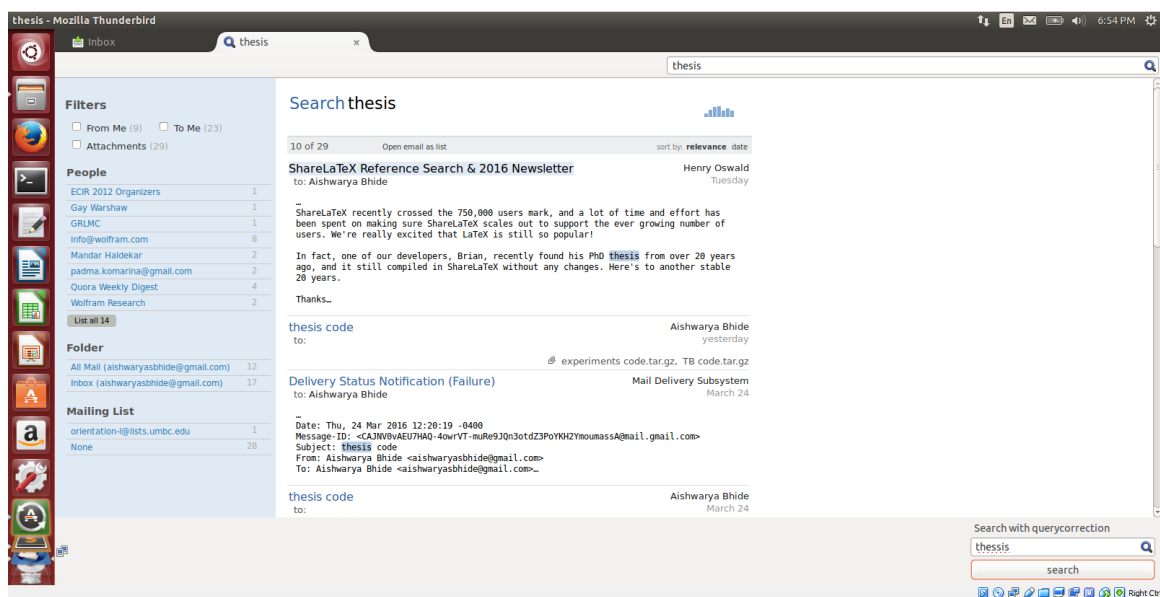


FIG. 3.5. Snapshot of output of add-on for query "thesis"

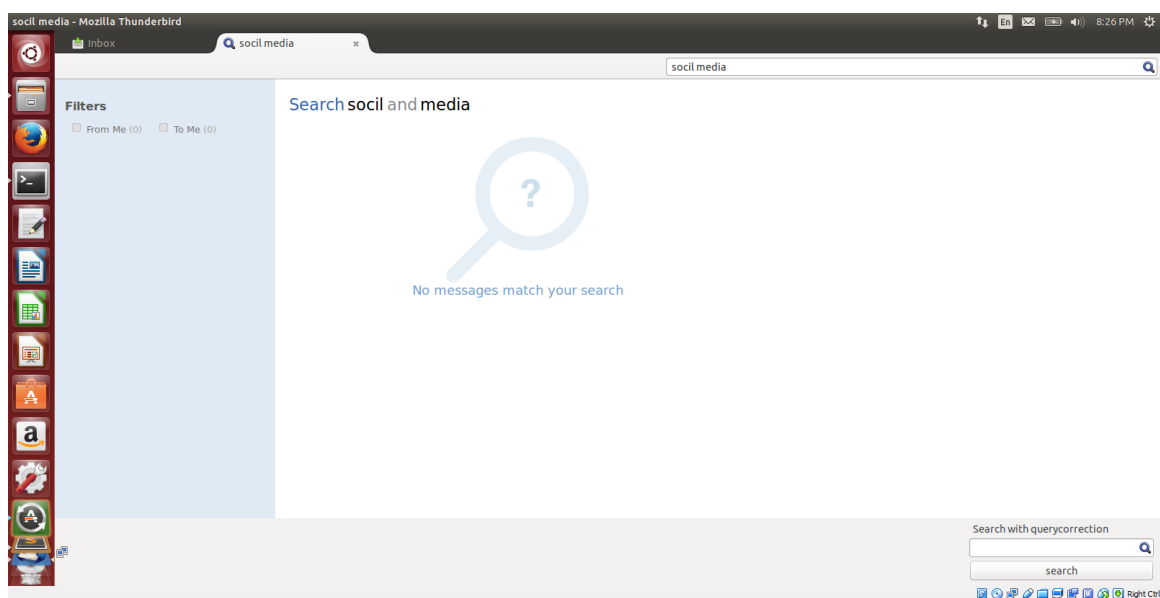


FIG. 3.6. Snapshot of output of Thunderbird global search for query "socil media"

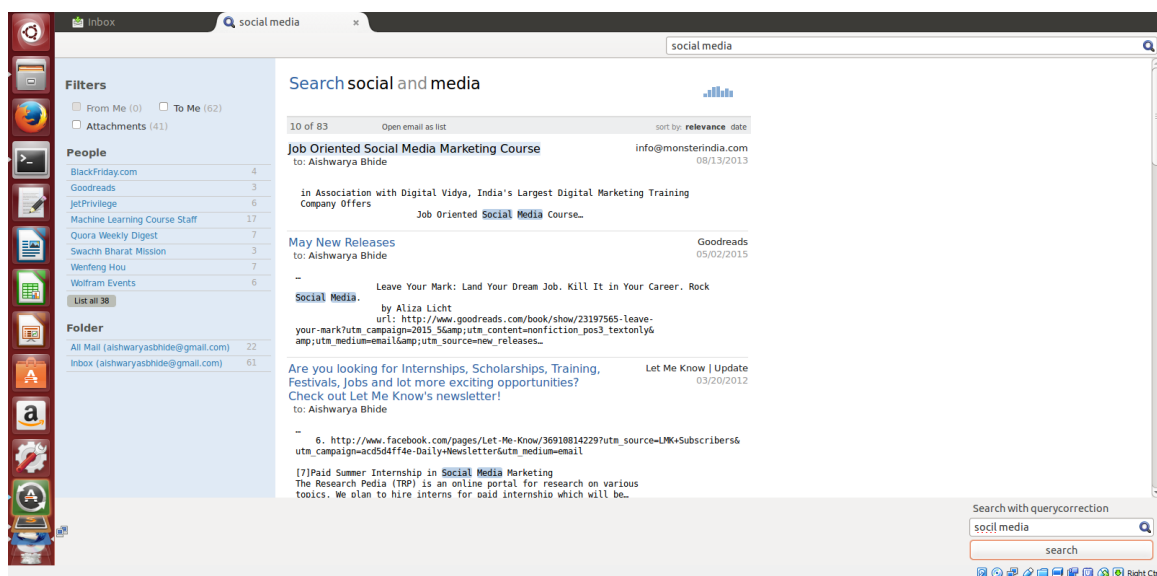


FIG. 3.7. Snapshot of output of add-on for query "social media"

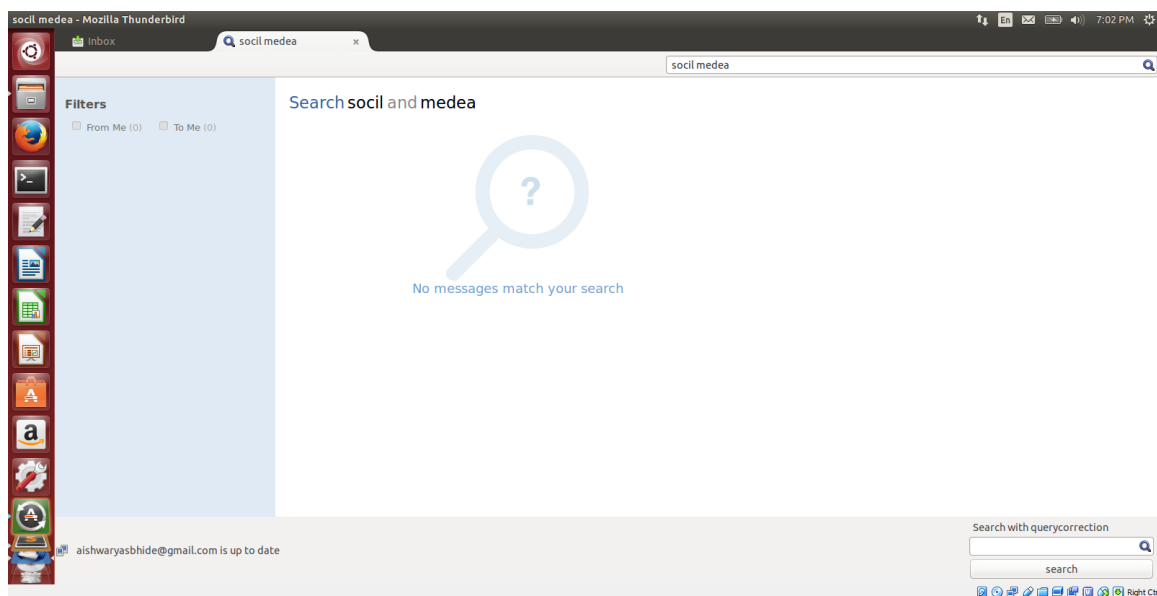


FIG. 3.8. Snapshot of output of Thunderbird global search for query "social media"

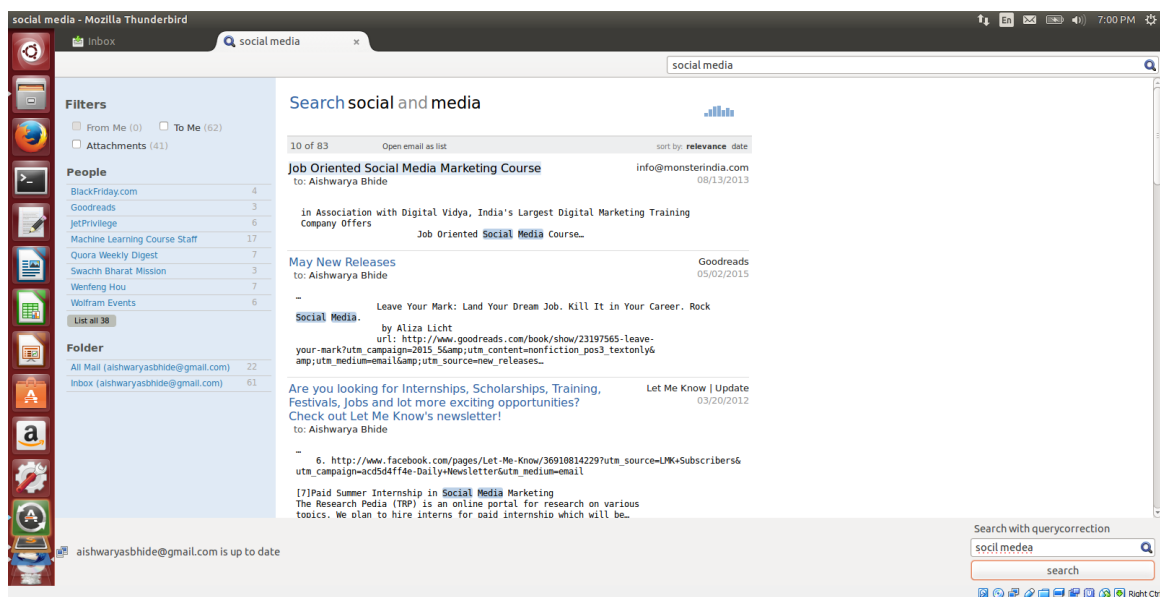


FIG. 3.9. Snapshot of output of add-on for query "social media"

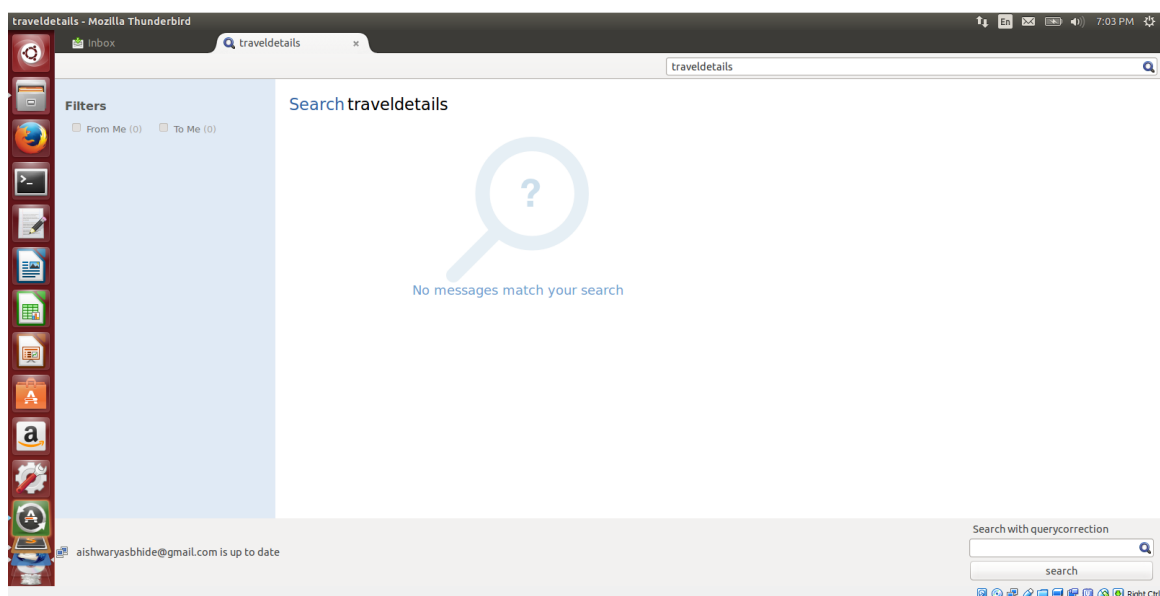


FIG. 3.10. Snapshot of output of add-on for query "traveldetails"

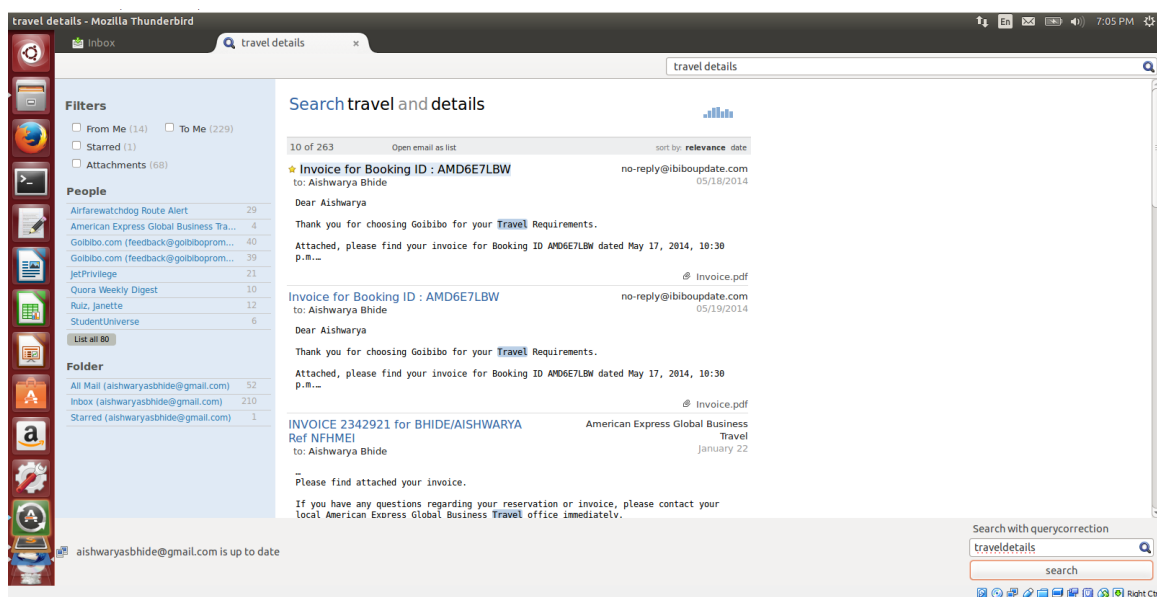


FIG. 3.11. Snapshot of output of add-on for query "traveldetails"

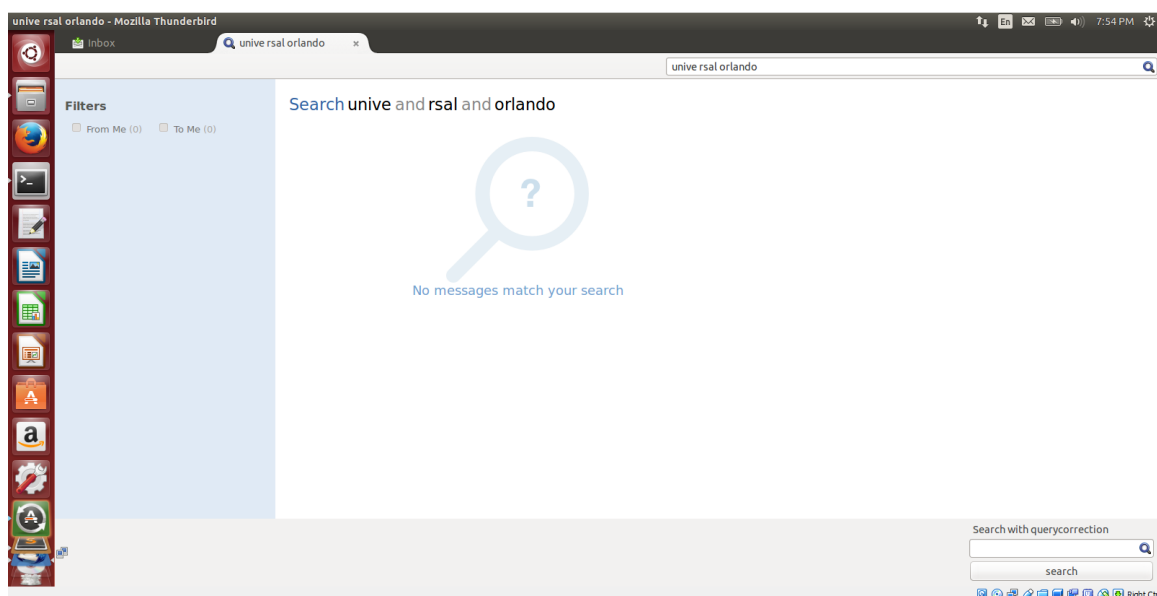


FIG. 3.12. Snapshot of output of add-on for query "unive rsal orlando"

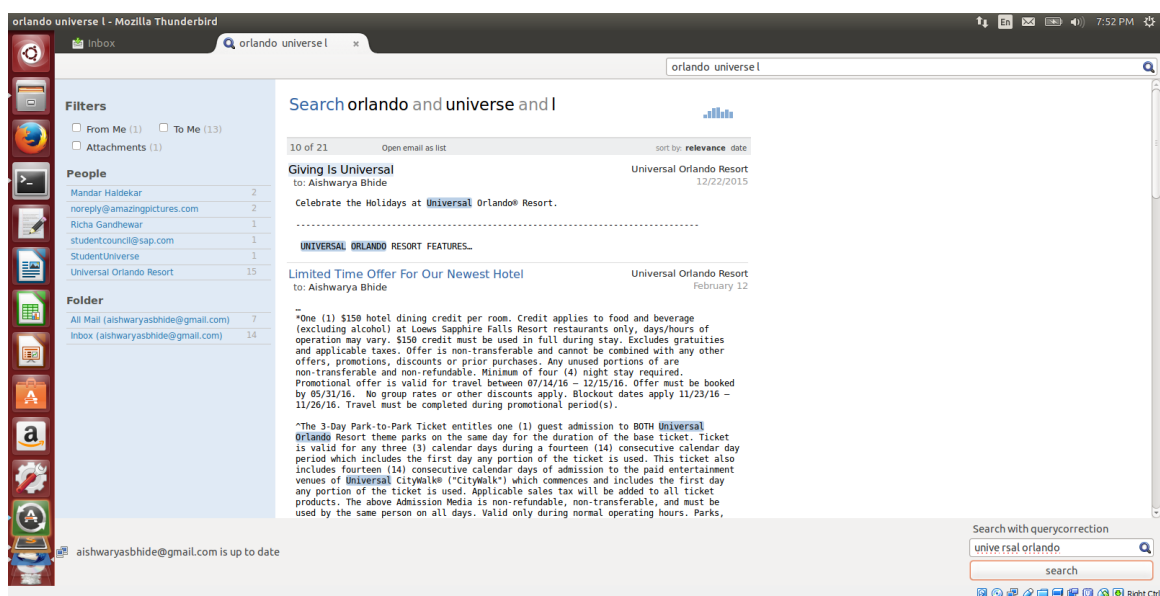


FIG. 3.13. Snapshot of output of add-on for query "unive rsal orlando"

Chapter 4

RETRIEVING RELATED EMAILS

Many times users want to refer to another email while reading a mail for details about the topic of current email. In this case, the user has to type the topic in search box and go through the search results. If the topic is very common, the search results may contain a large number of emails, only a few of which are related to what user is searching for. In most such cases, the related email user wants to refer to is from the same sender as current email.

This study implements the feature of showing related emails to current open email based on email content on the click of a button. The button is shown below the open email in status bar of Thunderbird. When user clicks the button, keywords are found from the email content using Part-of-Speech (POS) tags, term frequency (TF) and Inverse Document Frequency (IDF). The Python library `Topia.termextract` is used for finding most probable keywords based on POS tags and getting term frequencies. Document Frequency of keywords is found by querying Gloda database. The keywords may consist of one or more words. The keyword having top TF-IDF weight along with sender's name is used as search query.

Fig. 4.1 shows an open email for which user wants to search related emails.

Fig. 4.2 shows results of related emails for above email without using sender's name

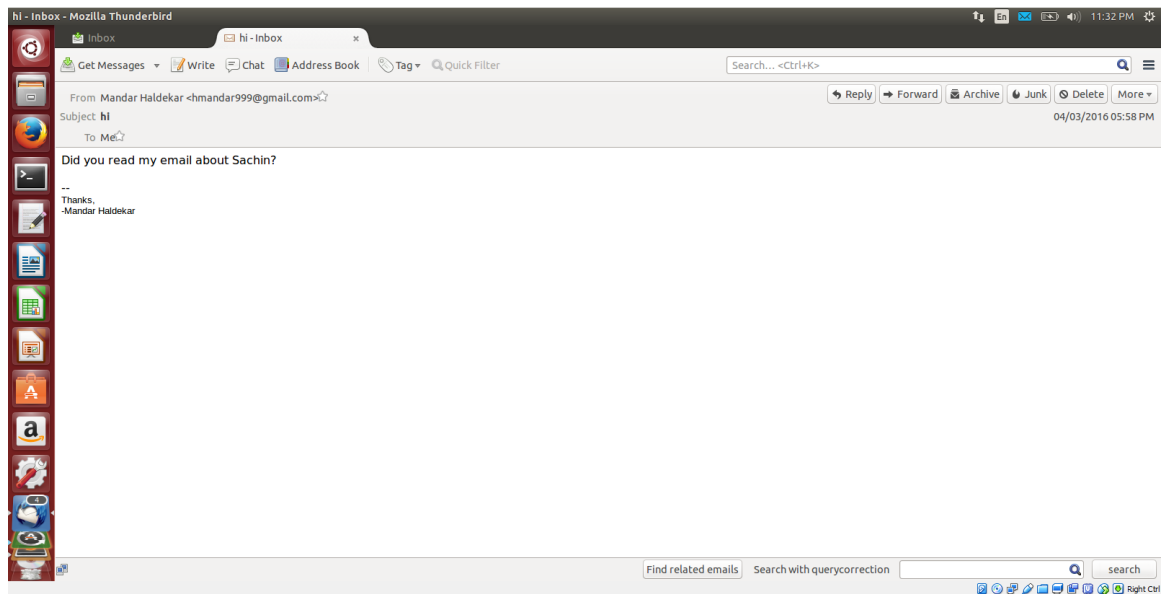


FIG. 4.1. Input email for related emails

on clicking 'Find related emails' button.

As seen in the results, there are many emails which contain the proper noun 'Sachin' which are not related to what user is searching for. Thus, the results don't readily show the expected email.

Fig. 4.3 shows the results of related emails for same email with sender's name in the query.

As seen in these results, this approach gives better results and shows the email that user is looking for with better accuracy.

Fig. 4.4 and 4.5 show another example of finding related emails for an email having a lot of text content.

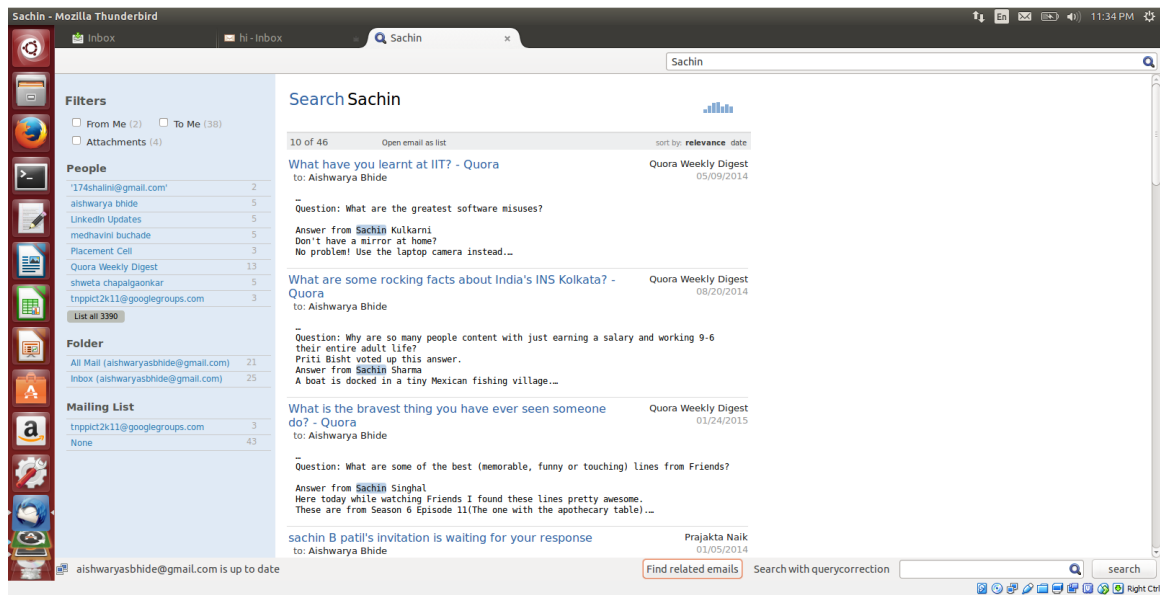


FIG. 4.2. Output without sender name in search query

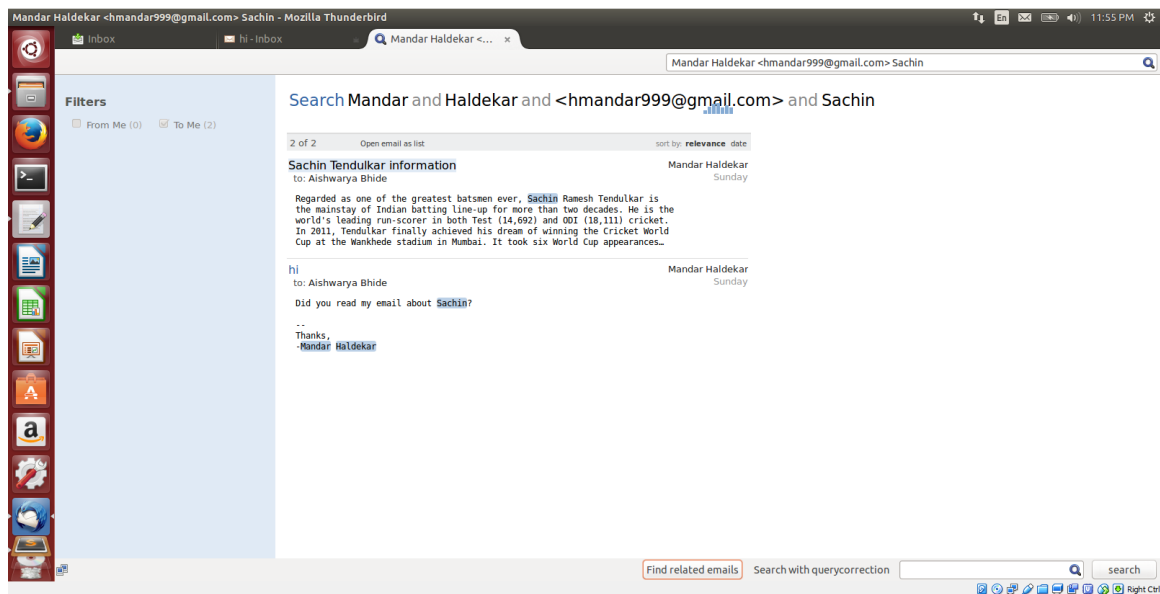


FIG. 4.3. Output with sender name in search query

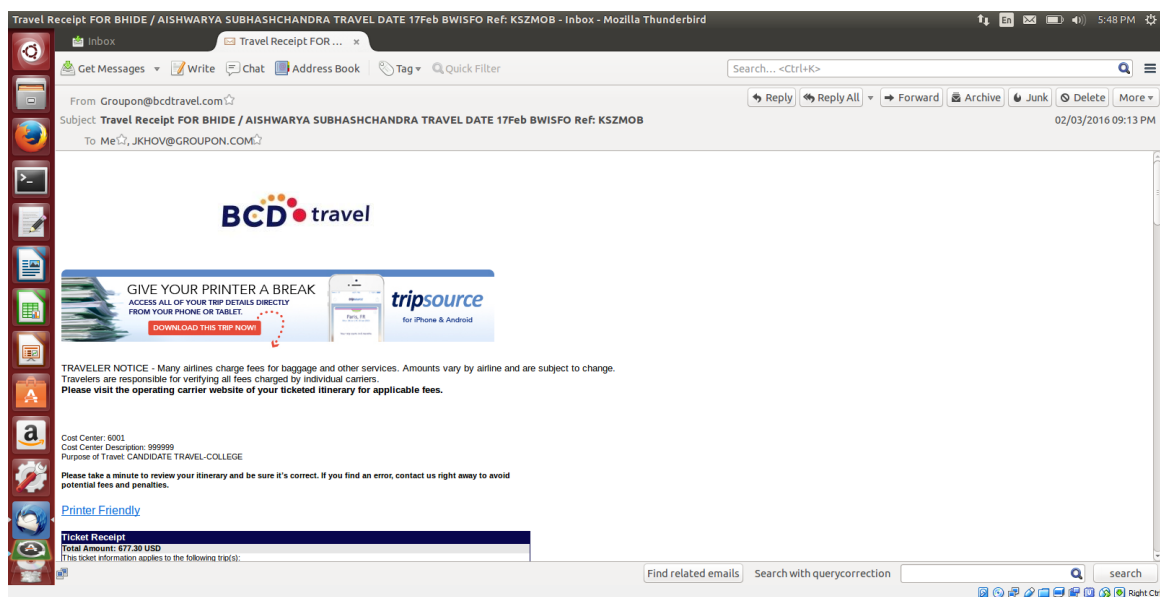


FIG. 4.4. Input email for related emails

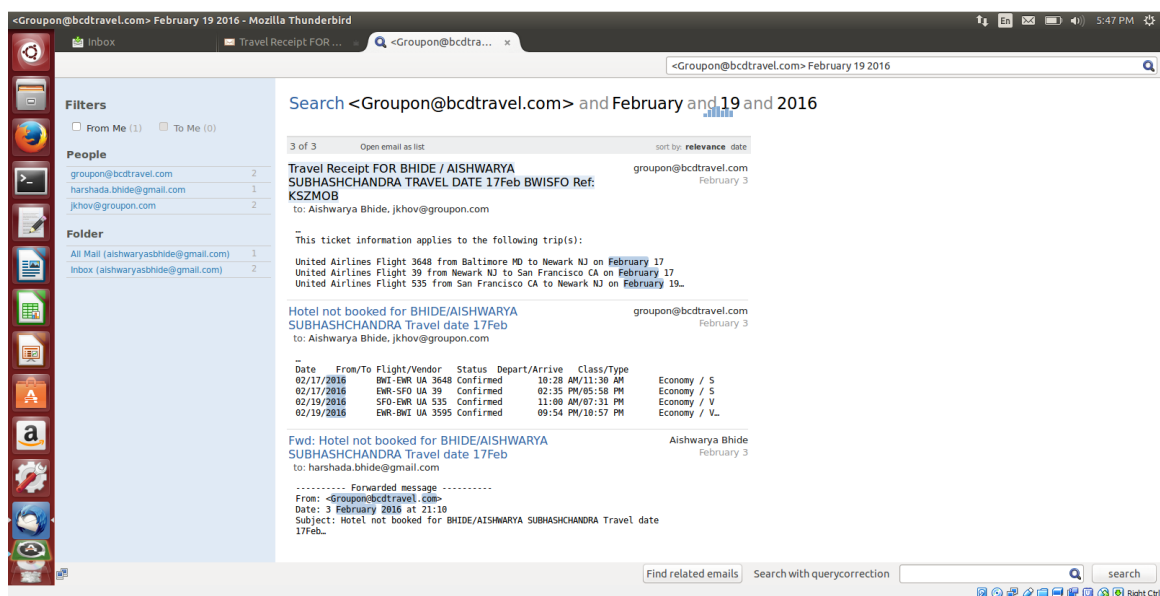


FIG. 4.5. Related emails for email in Fig. 4.4

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This study tries to enhance the search functionality of Thunderbird, an open source email client by providing the features of query auto-correction and displaying emails related to currently open email. Various experiments were performed to correct the text search queries. It was found that using machine learning algorithms did not give good results as different factors affected misspellings of words differently and predicting the correction in a word based on the observed misspellings of other words was rarely effective. Instead, making simplifying assumptions about the misspellings and using constant weights for each feature reduced the complexity and gave better results. Also, it was found that emails related to an open email could be found by using term extraction techniques and using extracted terms as search queries. Both of these enhancements would improve user's email search experience whether user's device is connected or not connected to internet.

5.2 Future Work

Performing synonym search i.e. searching for a word as well as its relevant synonyms, for the entered query would return better results. The query correction and search time could be improved by creating a persistent database index instead of using virtual tables.

A configuration option can be provided so that the user can choose whether to use virtual tables or persistent tables for index based on memory availability.

REFERENCES

- [1] Abhijeet Bhole, R. U. 2015. On correcting misspelled queries in email search. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*.
- [2] Buitinck, L.; Louppe, G.; Blondel, M.; Pedregosa, F.; Mueller, A.; Grisel, O.; Niculae, V.; Prettenhofer, P.; Gramfort, A.; Grobler, J.; Layton, R.; VanderPlas, J.; Joly, A.; Holt, B.; and Varoquaux, G. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 108–122.
- [3] Duan, H., and Hsu, B.-J. P. 2011. Online spelling correction for query completion. WWW 2011.
- [4] <http://www.cs.cmu.edu/einat/datasets.html>.
- [5] <https://www.sqlite.org/fts3.html#fts4aux>.
- [6] <https://pypi.python.org/pypi/python-levenshtein>.
- [7] <https://pypi.python.org/pypi/metaphone/0.4>.
- [8] Nicolas Ducheneaut, L. A. W. 2005. In search of coherence: A review of e-mail research. *HUMAN-COMPUTER INTERACTION* 20:11–48.
- [9] <http://www.nltk.org/>.
- [10] <http://norvig.com/ngrams/spell-errors.txt>.
- [11] Qing Chen, Mu Li, M. Z. 2007. Improving query spelling correction using web search results. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 181–189.

- [12] Silviu Cucerzan, E. B. 2004. Spelling correction as an iterative process that exploits the collective knowledge of web users. *EMNLP* 4:293–300.
- [13] <https://www.sqlite.org/spellfix1.html>.
- [14] <https://developer.mozilla.org/en-us/add-ons/thunderbird>.
- [15] <https://pypi.python.org/pypi/topia.termextract/>.
- [16] <https://pypi.python.org/pypi/whoosh/>.
- [17] <https://developer.mozilla.org/en-us/docs/mozilla/tech/xul>.
- [18] Zhuowei Bao, Benny Kimelfeld, Y. L. 2011. A graph approach to spelling correction in domain-centric search. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, 905–914.

