

A Case-Based Reasoning Approach to Email Response

James Morris

05-05-2017

Contents

1	Introduction	4
2	Background	4
2.1	Case-based reasoning	4
2.1.1	Textual case-based reasoning	5
2.2	Term frequency-inverse document frequency (Tf-idf)	5
2.3	Latent semantic analysis (LSA)	5
2.4	Previous work	6
3	Specification and Design	6
3.1	User requirements	6
3.2	Usability	7
3.3	User interface	7
3.4	Data flow	8
3.5	Design	9
3.5.1	EmailLoader	9
3.5.2	EmailSender	9
3.5.3	Classifier	9
3.5.4	Syntax parser	10
3.5.5	LSA	10
4	Implementation	10
4.1	Email	10
4.1.1	Load emails	10
4.1.2	Send email	11
4.2	Training data	11
4.3	Training data pre-processing	11
4.4	Vectorising	12
4.5	Classifier	13
4.6	Syntax parser	13
4.7	LSA	14
4.8	Challenges	15
4.8.1	Adapting whole emails	15
4.8.2	Question identification	15
4.8.3	Writing styles	16
4.8.4	GUI development	17
4.8.5	Measuring similarity/Creating vectors	17
5	Results and Evaluation	18
5.1	Testing	18
5.1.1	Parser testing	18
5.1.2	Case base testing	19
5.1.3	Lsa testing	19
5.1.4	Response generator testing	20

5.1.5	Manual testing	20
5.2	Evaluation	22
5.3	Issues	22
5.3.1	Context of conversation	22
5.3.2	Classification	22
6	Future Work	23
6.1	Check responses to store	23
6.2	Spelling correction	23
6.3	Feedback to user	24
6.4	Polish UI	25
6.5	Support for more emails	25
6.6	Paragraph vector	25
7	Conclusion	25
8	Reflection	27
	References	28

1 Introduction

Today email is one of the most prevalent forms of communication, spread across a many societies and social groups and particularly predominant in business for interaction whether it be between colleagues, customers or any other relevant parties. Even despite the rise of social media email has still managed to maintain its dominance in the market with 85% of citizens using the internet for email, whereas 65% have said to use it for social networking [1]. Considerable time is taken each day to respond to the vast amount of emails that could possibly be received - with "catching up on emails" being a staple in most peoples morning routine. Can you imagine the time that could be saved and improve productivity if there was a way to support this process and make progress towards full automation.

This project aims to explore case-based reasoning as an approach to provide some level of automation to email response. Due to the retrieval based concept of case-based reasoning, providing results to an open domain world would be impossible and therefore assumes a closed domain will be looked at. Due to the specific nature of some roles such as customer service, or in the case I have focused on - a school admissions tutor, there is usually a small target domain on which the emails will focus on. This allows a closed domain assumption to still provide reasonably good results with time.

2 Background

2.1 Case-based reasoning

Case-based reasoning, put in simple terms is the process of constructing solutions to unseen problems based on answers given to previous problems. The main feature of case-based reasoning is the case base, which consists of cases. Cases are made up of the past problem, the solution and any other information that the user sees fit to store. A common approach used is to store the case as a list of attributes. The process consists of four main parts:

- Retrieve - This is the process of selecting the most similar case or cases in the case base to new problem. Different approaches can be used at this stage, with different measures of similarity. It is up to the user to find the best that fits their application.
- Reuse - This is the process of reusing the selected case(s) to solve the problem.
- Revise - This is the process of adapting the reusable cases if necessary to fit the new problem.
- Retain - The proposed solution is then stored as a new case in the case base.

[16] One of the principles behind case-based reasoning is continual improvement. As it solves more problems and stores more cases its range of knowledge and quality of proposed solutions improves.

2.1.1 Textual case-based reasoning

A small subset of case-based reasoning which is particularly applicable to the case of emails is textual case-based reasoning. This applies the same main components as case-based reasoning, with the added complexity that all of the cases are in a textual format [20].

2.2 Term frequency-inverse document frequency (Tf-idf)

This is a weight measuring how important a word is to a corpus. The weight value will be:

- Largest when a word is in few documents but appears many times.
- Lower when a word is in many documents or appears few times.
- Smallest when a word is contained in nearly every document (will be 0 if in every document).

[4]

The calculation is as follows:

$tf(t) = \text{Number of times } t \text{ appears in the document} / \text{Total number of terms in the document}$

$idf(t) = \log(\text{Total number of documents} / \text{Number of documents containing } t)$

$$tf-idf(t) = tf(t) * idf(t) [17]$$

If there were 20 documents and the word "hello" appears in 4 of them and the document we were looking had 30 terms and "hello" appears once the tf-idf weight would be:

$$\frac{1}{30} * \log\left(\frac{20}{4}\right) = 0.023$$

A document vector can be created using this measurement with a term for each word in the vocabulary and its corresponding tf-idf weight.

2.3 Latent semantic analysis (LSA)

Latent semantic analysis is a way to try and understand the meaning behind a given piece of text. It is used to analyse the relationships between a set of terms. First a term-document matrix A is created using the bag-of-words model. Then singular value decomposition (SVD) is performed. If we say $B = A^T A$ and

$C = AA^T$ then a SVD on A is equal to:

$$A = S\Sigma U^T$$

Where:

S is the matrix of the eigenvectors of B

U is the matrix of the eigenvectors of C

Σ is the diagonal matrix of the singular values obtained as square roots of the eigenvalues of B. [18]

2.4 Previous work

An approach to email response using case-based reasoning was carried out by Lamontagne and Lapalme [12]. This focused on the investor relations domain. In this approach whole emails are stored which are then retrieved and reused by removing unrelated information in the selected response. In the system I have proposed the need for this step is omitted. As smaller questions and not whole cases are stored, all information stored as the response will be relevant to the question being asked. The quality of retrieval should also be improved in my system. Matching email to email when multiple questions could be asked in each, will result in emails containing small relevant passages, making similarity hard to identify.

Google have proposed a solution - Smart Reply - that is currently in use in Gmail inbox and 10% of mobile replies are being assisted by this system [2]. This approach focuses on short emails due to a study finding roughly 25% of replies having 20 terms or less. To perform this they utilise the sequence to sequence learning framework and long short term memory networks to predict sequences of text. Here they present multiple choices for your response including both positive and negative views. Obviously my resources are limited compared to Google, and possible approaches they can take are greater due to them having 238 million training examples, but I can still present a key difference that may be an advantage in some situations. It seems the system can only handle short responses and from what I can see can handle one query. Although my system requires storage of cases it presents a trade off between the types of input it can handle.

3 Specification and Design

3.1 User requirements

- System must allow the user to load emails from their desired mailbox.
- System must allow the user to send their response to a given email.

- System must be able to identify similar questions to the given input.
- System must be able to identify if a sentence is in the form of a question.
- System must be able to extract the name of the sender.
- System must be able to identify the topic of a given sentence.
- System must be able to take an email as input and generate a suitable response.
- System must allow the user to type in a desired email.
- System must allow the user to edit the response.
- System must allow the user to provide a response where the generator was unsuccessful.
- System must allow the user to save responses to a given question.
- System must allow the user to save the topic of a sentence.
- System must allow the user to save the similarity threshold value.
- System must allow the user to edit their email signature.

3.2 Usability

The best case usability of the system would be to have a button added to your existing email client to generate a response for the selected message. This could be done in the means of an add on to your browser to work with your email client. If this route was taken an add on would have to be created for each potential browser and each email client, which would be very time consuming. This is why I have chosen to create a standalone system. To improve familiarity I have tried to keep the design and functionality as close to what we usually see with the most common email clients.

3.3 User interface

I opted for a simple user interface design encompassing two text areas for the input and response emails, a list view to load emails into and buttons to login and generate a response. Settings and options to train will exist in menu items. When a response cannot be found for a given question; text inputs and dropdown fields will be dynamically added and removed when saved.

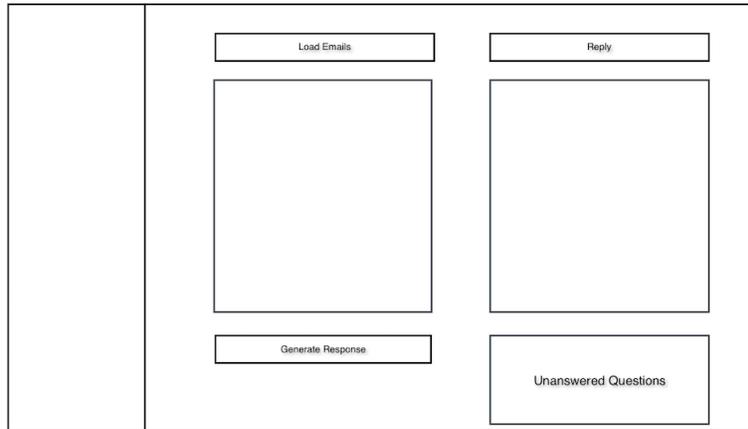


Figure 1: Wireframe of main screen

3.4 Data flow

- Username and password is passed to EmailLoader and EmailSender.
- List of loaded emails is returned to the users.
- Email is selected from list or input by user.
- Case base is loaded into ResponseGenerator.
- Input is tokenised into sentences.
- Each sentence is classified, topic and probability is returned.
- Each sentence is parsed - Penn Treebank tagged and NER is returned.
- Each sentence is classified as sentence or statement.
- Questions are combined with same topic statements.
- Each new piece of text is sent to Lsa.
- Most similar responses and unanswered questions are returned.
- Most similar responses are combined and returned to the user along with the unanswered questions.

3.5 Design

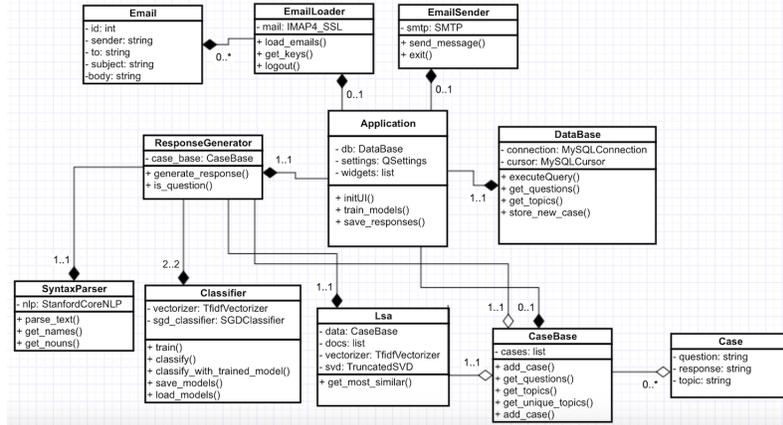


Figure 2: UML class diagram of the system

3.5.1 EmailLoader

EmailLoader is responsible loading emails from a given mailbox into the program. It makes use of the Internet Message Access Protocol (IMAP) to retrieve mail from a given email server. The received email is converted to a more friendly defined object for use within the system.

3.5.2 EmailSender

EmailSender handles the sending of the response message. It makes use of the Simple Mail Transfer Protocol (SMTP) to send to a given server.

3.5.3 Classifier

The classifier implements a Stochastic Gradient Descent classifier using a log loss function. It has two purposes within the system. The first classifier is used to label a given sentence with a topic that it relates to. This is trained using the cases that are stored in the case base. Depending on the amount of training data you have, there is a possibility that a new sentence does not fit into the set of labels. In this case I am looking at the highest label probability and checking it meets a certain threshold value. If this is not met it is classed as no topic given. This would then require the user the evaluate the sentence and provide a new label. If enough training data is given to cover all topics this threshold would not be necessary. The second is used to determine if a given sentence is in the form of a question, if this cannot be determined from the syntax parser.

3.5.4 Syntax parser

Syntax parser uses the Stanford CoreNLP, using the parser and named entity recognition tools. The parser is used as the first approach to determine if a given sentence is in the form of a question, by looking at the Penn Treebank tags given. The named entity recognition is used to extract the name of the sender of the received email, to address them in the generated response.

3.5.5 LSA

This class performs latent semantic analysis to find the most similar question(s) in the case base to the ones extracted from the input email. This is achieved by creating a tf-idf vector of the input question and then performing singular value decomposition on this vector. I am then using cosine similarity to analyse how closely related the input question is to each question in the case base. A threshold value - that is editable through settings - is used as minimum level of similarity that the most similar question has to reach. If this is not reached it is assumed no question exists within the case base that could reasonably answer the question. In this circumstance user input would be required to provide a suitable response.

4 Implementation

4.1 Email

4.1.1 Load emails

To load emails I am making use of python's built in imaplib library. To connect to the host I am using the IMAP4 subclass IMAP4_SSL. This adds the security of connecting over an SSL encrypted socket. Then logging in with the login function. To search for mail a mailbox needs to be selected to look through, in most cases this will be the inbox.

```
def __init__(self, host, port, mailbox, username, password):
    self.mail = imaplib.IMAP4_SSL(host, port)
    self.mail.login(username, password)
    self.mail.select(mailbox=mailbox, readonly=True)
```

Now we need to loop over the mailbox to fetch all of the emails. To do this we first need to get all of keys associated with each email. The imaplib has a search function that we can use to search through the mailbox using a specified search criterion. In our case where we want to grab every email we can just specify "ALL". To then have the most recent emails first we must reverse the list. [9]

```
def get_keys(self):
    typ, data = self.mail.search(None, 'ALL')
    return data[0].split()[::-1]
```

Then we can loop over each key and fetch email for the given key specifying the parts of the email we wish to return. The returned data is not very friendly to handle so we can use the email module and function `message_from_bytes` to help with this.

```
def load_emails(self):
    messages = []
    keys = self.get_keys()
    for key in keys[:40]:
        typ, data = self.mail.fetch(key, '(RFC822)')
        raw_email = data[0][1]
        message = email.message_from_bytes(raw_email)
        messages.append(Email(message))
    return messages
```

4.1.2 Send email

For sending emails python has another built in library that we can use - `smtplib`. Specifying the host and port we want to connect to. To add a level of security we can use `starttls` to put the connection in TLS mode. Every command that follows this will be encrypted [10].

```
def __init__(self, host, port, username, password):
    self.smtp = smtplib.SMTP(host, port)
    self.smtp.starttls()
    self.smtp.ehlo()
    self.smtp.login(username, password)
```

4.2 Training data

The training data for the topic classifier is the data we have stored in the case base, only the response is omitted. A list of the questions and a separate list of the corresponding topics is passed to the classifier.

The training data for the questions classifier is stored in a `.csv` file which comprises of a sentence followed by a label in this case either "Question" or "Statement". Below shows a possible example:

```
I have attached all my documents with this email.,Statement
I want to know whether I am eligible to enrol directly with final year.,Question
```

4.3 Training data pre-processing

Before any of the models are trained including both vectoriser and classifier the training data is pre processed to improve the end performance. The first step

is to remove punctuation as after tokenising words with punctuation will be treated as different by the classifier even though the meaning is the same for example:

```
jump
jump.
jump,
```

Most words can have endings based on tense or if they are plural. Essentially this is not going to have much effect on the meaning of the sentence but will have an unnecessary effect when measuring similarity between two sentences.

```
jump
jumping
jumps
```

To fix this I stem the words using the porter stemmer as described earlier. This helps reducing the chances that the classifier is encountering a new word that it has not been trained on and reduces the probability space [14].

```
text = [stemmer.stem(word) for word in nltk.word_tokenize(text.translate(punct).lower())]
text = ' '.join(text)
```

4.4 Vectorising

In both cases of the classifier and latent semantic analysis, operations cannot be performed on the data in its current state. To solve this the sentences must be transformed to the vector space. There are a few ways to do this such as the standard bag-of-words model. This creates a vector with dimension $|V|$ where V is the vocabulary, where each value in the vector represents the number of occurrences of a given word in the document [4]. In this case common, unimportant words such as "a", "and", "the" are always likely to have the highest frequency and important words have little effect on the similarity. To solve this problem I have decided to use tf-idf as weighting for each word. This improves the model by giving a higher value to words that are more important and play a more significant part in the meaning of the document, and minimises the effect of frequently occurring terms [19]. To perform this I am using scikit learn `TfidfVectorizer`. To this I am passing a list of stop words - these are commonly used words, like mentioned above that add no value to the meaning of a sentence that I have chosen to ignore [7]. Then we can train the vectoriser and return the transformed data using `fit_transform`.

```
self.vectorizer = TfidfVectorizer(stop_words=stop_words)
tfidf_train = self.vectorizer.fit_transform(data)
```

4.5 Classifier

I have used a classifier for a few key features that support some of the key functionality. For the classifier I have used sci-kit learn `SGDClassifier`. This is a linear model using stochastic gradient descent learning [8]. I have chosen to use this particular classifier as it is one of the only ones to provide me with functionality to access the probability of a given class label. Given the specified loss function this can either implement a support vector machine or probabilistic classifier. In my case where I need the probability value for the given class I have decided to use the log loss function. Then we need to train the model using the fit function. This takes two lists - the training data and the target values. In our case the data will be the questions in the case base to identify topics or the question and statement data to identify a question. Before we can pass this data to the function we must vectorise it. To do this I am using sci-kit learn `TfidfVectorizer` as mentioned above. The resulting transformed data is what we pass to the classifier.

```
self.sgd_classifier = linear_model.SGDClassifier(loss='log',alpha=0.0001)
self.sgd_classifier = self.svm_classifier.fit(tfidf_train, labels)
```

As the amount of training data increases the time to train the models increases. If we were to train the models each time we choose to generate a reply the system will take a heavy performance hit. To fix this we can use model persistence. This allows us to use a previously trained model without having to retrain. To save the trained models I had to decide between using pickle or sci-kit learn `joblib`. I ended going with `joblib` as it is more efficient when dealing with objects that internally use large numpy arrays as is the case with the models I am using [6].

```
joblib.dump(self.sgd_classifier, classifier_filename)
```

```
self.sgd_classifier = joblib.load(classifier_filename)
```

Before being able to classify we must first transform the input text using the vectoriser model we trained earlier. Then we can use the classifier predict function to predict the class. In some cases where we do not have enough training data, the class label that should be given to a piece of text does not yet exist. To check for this I am checking if the probability associated with the class label meets a threshold. The `predict_proba` function returns the probability for each label in the data, so the value I need to check against is the max value in that list.

4.6 Syntax parser

The parser is used as a first step to identify questions and to perform named entity recognition on each sentence. All of these tools are available through the

Stanford Core NLP suite. As the library is written in Java this posed a problem as I am using Python. Fortunately as part of the package there is a server that you can run to accept POST requests. I used a handy library `py-corenlp` that handles these requests nicely. To run the server the command below is executed:

```
java -mx4g -cp "*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9000 -timeout 15000 [11]
```

To create the library object the url of the running server must be passed.

```
self.nlp = StanfordCoreNLP('http://localhost:9000')
```

Now we can use the `annotate` function passing the sentence and the list of annotators. In our case we need "parse" and "ner".

```
output = self.nlp.annotate(text, properties={'annotators': 'parse, ner', 'outputFormat': 'json'})
return output['sentences'][0]
```

From "ner" we get a list of json objects for each word in the sentence with a "ner" name field to indicate the entity value or 0 for no entity. From this list I then needed to extract the full name of any persons. To do this I just iterated over the list adding consecutive words tagged as "PERSON" to a list to enable me to easily extract the first name.

```
sentence_tokens = parsed_sentence['tokens']
names = []
name = []
sentence_length = len(sentence_tokens)
for i in range(sentence_length):
    if sentence_tokens[i]['ner'] == 'PERSON':
        name.append(sentence_tokens[i]['word'])
        if i == sentence_length - 1:
            names.append(name)
    else:
        if name:
            names.append(name)
            name = []
return names
```

4.7 LSA

Before we vectorise the text to improve the results I will use the pre processing steps described above.

Here again I am using the `TfidfVectorizer`.

A key phase of latent semantic analysis is singular value decomposition, to perform this I am using sci-kit learn TruncatedSVD.

```
self.__svd = TruncatedSVD(n_components=n_components)
textlsa = self.__svd.fit_transform(textfidf)
```

The similarity measure I used is the cosine similarity, this is most commonly used in information retrieval.

```
sim = cosine_similarity(textlsa[0:1], textlsa[1:len(textlsa)])
```

This measures the similarity between two vectors. More precisely it is the angle between the vectors. The calculation is as follows:

The dot product of two vectors $a = (a_1, a_2, a_3, \dots, a_n)$, $b = (b_1, b_2, b_3, \dots, b_n)$

$$a \cdot b = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3 + \dots + a_n b_n$$

The magnitude of a vector a

$$\|a\| = \sqrt{(a_1)^2 + (a_2)^2 + (a_3)^2 + \dots + (a_n)^2}$$

$$\cos = \frac{a \cdot b}{\|a\| \|b\|} \quad [5]$$

4.8 Challenges

4.8.1 Adapting whole emails

Initially my idea was to store whole emails and match the whole input to the most similar. If there was a guarantee that each email was about one topic and asking one question then this approach would have been satisfactory, but this is no guarantee. A key part of case-based reasoning is the adaptation phase. In the case of textual case-based reasoning this is even more tricky. Unlike case-based reasoning where adaptation has been researched, after doing considerable research on textual it seems adaptation has been focused on structural cases, I therefore decided that this was not the most appropriate choice to take. As part of this research I came across a few other approaches that others have taken instead of trying to adapt when using textual case-based reasoning. The approach I decided on is composition as seen above which comprises of filling the case base with questions and responses and matching questions in the input to them.

4.8.2 Question identification

The part of an email that dignifies a response is generally the fact that a question has been asked. Using the approach of composition the identification of

what sentences are in the form of a question is a key aspect required to allow the system to start working on response. Due to the different ways in which people ask questions this is not a straight forward task. A simple way to check would just be the presence of a question mark, but there is no guarantee that a sender will use correct grammar or punctuation, so therefore the focus needs to be on the words written. Regarding the words another simple approach would be to check if the sentence starts with what/why/when, but again questions can be written in many ways with different word ordering, so this will only pick up a fraction of cases.

After going with the Stanford parser and checking the penn treebank tags, there are cases where SQ and SBARQ do not cover all questions. In particular indirect questions are picked up in SBAR tags, but thats not all SBAR tags pick up. In many circumstances sentences that are not indirect questions are labeled as SBAR if we look at what the tag represents we can see why:

"SBAR is used for relative clauses and sub ordinate clauses including indirect questions." [3]

As I do not have the required knowledge to understand the technical structure of questions it was difficult for me to derive a way to distinguish between a question or statement that is tagged as SBAR. After doing research in the area I am not sure if it is even possible to determine this based on just the tags. To get around this I trained a classifier to handle the cases where only SBAR exists to determine if the sentence is a statement or question.

4.8.3 Writing styles

My initial way of thinking was to tokenise the input email into sentences, just take the questions and find the most similar in the case base, disregarding all of the statements. However because of the different styles of writing each individual possesses this has many flaws. if we look at the example below:

I am studying maths, physics and chemistry, do you accept these A-levels for entry?

If written in this way all required information will be picked up by using my first approach. Another way of writing this could be:

I am studying maths, physics and chemistry. Do you accept these A-levels for entry?

The same as the first but split into two sentences. Now if we just take the question, we are missing crucial information that is necessary to successfully answer the question. A simple solution would be to take the question and the sentence

before it. But if we have:

I am studying maths, physics and chemistry. I am from Cardiff. Do you accept the A-levels mentioned for entry?

or

Do you accept these A-levels for entry? I am studying maths, physics and chemistry.

In a lot of circumstances the sentence before could be completely unrelated. With the addition of more questions and statements adding to the complexity of an email there is no pattern of ordering that exists for all cases. Therefore given all of the separate sentences of an email I needed a way to work out what statements, if any, are related and required to be able to answer each of the questions being asked. To do this again I have used a classifier to determine the topic of all the sentences in the input and then match questions to the sentences where the topics are the same.

4.8.4 GUI development

Due to my inexperience with creating graphical user interfaces using python, or any language for that matter it took longer than I initially thought to create the user interface and the workings behind it. First I had to decide what GUI package to use. I had briefly used tkinter before and not really liked it so I decided to use PyQt as it seemed to be another popular choice that had good documentation. To gradually build the interface I utilised tutorials and examples online to gain knowledge of the widgets and layouts available within the package and how to use them correctly.

4.8.5 Measuring similarity/Creating vectors

Measuring the similarity between the input and case base is crucial to the quality of the response that is given. A simple method I initially went with was to use the classic bag-of-words model, but as the number of words that are added to the corpus is increased the vectors move towards having a majority of zero entries, which tends towards making all vectors appear similar to each other even though in real terms of the sentence they are not. All words also have an equal value of weight in terms of measuring similarity, even words that do not have an effect on the meaning of the sentence, which is clearly not optimal. To improve this I decided to use latent semantic analysis, this uses tf-idf vectors as described above.

5 Results and Evaluation

5.1 Testing

To perform most of my testing I have written a set of unit tests using the Python unittest module. Having these tests allows me to know what parts of the system are still working after changes have been made. The advantage of this is I can make all of these checks without having to manually run through the system, which can be very time consuming.

To test the accuracy of my classifier I have used a small test set of sentences and compared the predicted label to the expected label.

```
classifier.load_models('question_classifier_model.pkl', 'question_vectorizer_model.pkl')
for sentence in test_sentences:
    pred_class, prob = classifier.classify(sentence)
    predictions.append(pred_class)

accuracy = metrics.accuracy_score(labels_test, predictions)
```

The accuracy of the questions classifier was 0.7

The question classifier was trained on a set of 38 statements and 15 questions. As you can see the amount of training data is quite small here and therefore the accuracy of the classifier should not be taken to seriously.

5.1.1 Parser testing

The Stanford parser itself I am assuming testing is being handled on their end so felt there was no need to test the responses I am receiving from it. What I am testing here are the functions I have written to extract data from the responses I am receiving.

```
get_names
```

This test verifies that the get_names function is correctly retrieves the full names from a sentence returned from the parser. The test uses different scenarios to make sure all bases are covered and that all names will always be found. This includes:

```
Bob Ford is my name.
This sentence has no name.
Lilly Nixon met with Brooke Carey for a coffee.
Do you know Rosie Farrell?
```

Each of these covers different situations including; one name at the start, no name at all, two names and one name at the end. If any other scenarios are

brought up they can easily be added. This test covers the requirement must be able to extract the name of the sender.

```
for sentence in sentences:
    text = self.parser.parse_text(sentence)
    names.append(self.parser.get_names(text))

self.assertEqual(names[0], ["Bob", "Ford"], 'Names are incorrect')
```

5.1.2 Case base testing

The case base testing focuses on making sure the functions are retrieving the correct information from the cases. First a case base has to be created with cases added that we can retrieve information from. It is unnecessary here to retrieve values from the database as this is not what is being tested.

get_topics

This test makes sure the correct list of all topics is returned from the case base.

get_unique_topics

This test makes sure a correct list of unique topics is returned from the case base.

```
topics = self.case_base.get_topics()
required_topics = ["Qualifications", "Qualifications", "Course Information", "Course Equipment"]
self.assertEqual(topics, required_topics, 'Topics are incorrect')
```

5.1.3 Lsa testing

The Lsa testing makes sure that the correct question is being selected as the most similar and the correct response is being returned. Again here a case base must be created to measure similarity against.

get_most_similar

This test verifies that the correct most similar question is matched and the correct response is returned.

get_most_similar_threshold

This test is the same as above apart from it is checking that if the threshold is not met then no response is returned.

These tests cover the requirement must be able to identify similar questions to the given input.

```
text = "Do you accept Chinese as an A-level subject"
most_similar_response = self.lsa.get_most_similar(self.case_base, text, 0.7, 24)
self.assertEqual(most_similar_response, "Yes, we accept A level Chinese.", 'Wrong text selected')
```

5.1.4 Response generator testing

Testing of the generate_response method I believe is unnecessary here as this function mainly makes call to functions that have already been tested. I have therefore just written a test for is_question.

is_question

This test is to verify that the function is correctly identifying a statement from a question. Multiple tests have been written here to test for multiple types of questions and statements. The benefit of doing this is it will enable us to see what types of questions are not being correctly identified which will allow us to make improvements to the classifier more efficiently. This test covers the requirement must be able to identify if a sentence is in the form of a question.

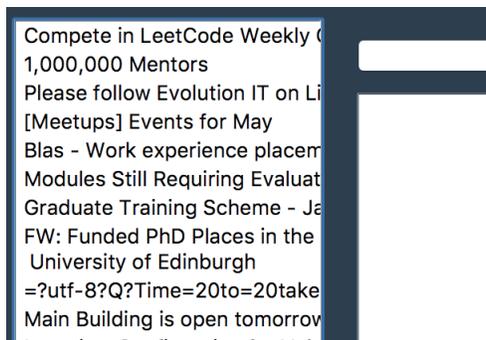
```
question = "Could you outline the different modules that are available."
parsed_question = self.parser.parse_text(question)
is_question = self.generator.is_question(question, parsed_question, self.question_classifier)
self.assertEqual(is_question, True, 'Sentence is a question')
```

5.1.5 Manual testing

Obviously in this case parts of the system cannot be tested using automation such as the quality of the responses being generated. Below are tests carried out manually and focus on requirements that were not easy to check using automated unit testing.

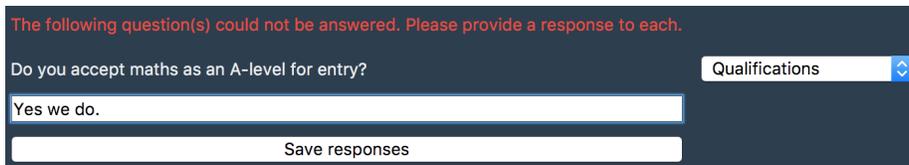
System must allow the user to load emails from their desired mailbox.

Below shows screenshots of the login dialog and emails loaded into the list view after a successful login attempt.



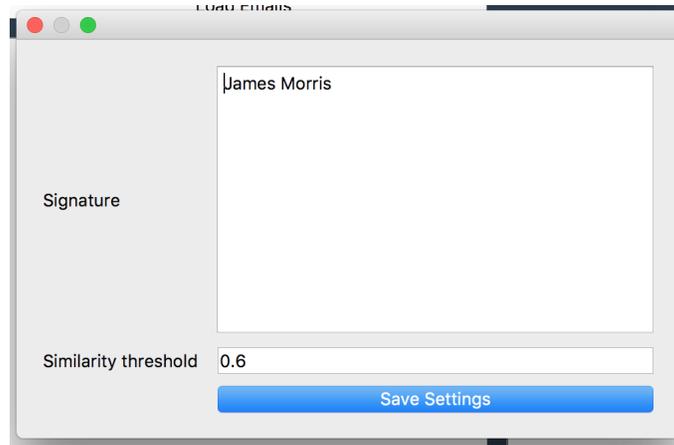
System must allow the user to save responses to a given question.
System must allow the user to save the topic of a sentence.

Below shows a screenshot of an unanswered question being displayed to the user for input.



System must allow the user to save the similarity threshold value.
System must allow the user to edit their email signature.

Below is a screenshot of the settings dialog that allows the user to save a signature and similarity threshold.



5.2 Evaluation

From the tests carried out and the results of all the tests passing we could say that the project has been a success. Although all of the requirements have been met, what is more difficult to measure through the ability to pass a requirement is the systems capabilities to provide responses to a wide range of emails. Yes right now it will identify a statement as a question in some cases, which will incorrectly ask the user to provide a response for it. This could add incorrect statements into the case base and pollute it with bad data while the case is built up. To combat this problem I have suggested below that checkboxes should be added to allow the user to select the responses they wish to store. A key characteristic of the system is it should improve as it is given more data. The question classifier should identify questions with better accuracy, the topic classifier should identify the topic of a sentence with better accuracy and the more questions in the case bases the more responses it could retrieve without requiring user input. As I do not have this data it is hard to say how well the system would perform.

Even if we had this extra data the system in its current state will still make mistakes. This is due to the issues described below, although the issue of classification could be solved by having extra data.

So does it help with email automation? At this stage with the amount of data the system has and the amount of emails I have to test it is hard to say how well the system works as a tool to automate email response. Only through extended use to provide more data could we begin to truly given an honest answer to this question.

5.3 Issues

5.3.1 Context of conversation

One problem that the system has is it is not aware of the context of the email conversation, as in it is not aware of any emails in the chain before the one it has just received. This is fine for the first email sent as nothing has come before it so no further information apart from what is in the current email is required. When an email is referencing the previous email sent the information being referenced is sometimes critical to being able to answer the question. For example below:

Thank you for your assistance on this matter. Could you please clarify which documents you need from me and I will send them over to you ASAP.

The response given and stored for this was written when the context was known and full knowledge of what documents were required was known. It would be possible for a similar email to be received with different context of what documents are required and in this case the wrong response would be given.

5.3.2 Classification

Initially my original assumption was whole emails would be stored and a single most similar email would be picked and completely adapted to an appropriate response. This entailed an assumption that an email would be about one topic and asking a singular question response. I realised this was not the correct thinking as I could not have any assumption really about what an email could contain, only something that would dignify a response. Then I moved to realise adapting just one email is likely to be impossible as not all the relevant information is certain to be contained in the stored response. The similarity between two text passages would also likely be less accurate and harder to determine. I then moved to retrieving multiple emails and adapting them together to form a response. Again here identification of the relevant pieces was difficult and providing the user with pieces that needed their input was tricky. Classification here has no guarantee that even if the topic is the same question is not guaranteed to be the same. Similarity is hard to measure as there could be additional text that provides no importance in the grand scheme to the email but will impact the similarity, so removal of this was key. All of the above lead me towards storing the individual questions. Here the classification matching is improved over the case above as there is a good chance they will relate to the same thing. Of course this will not work 100 percent of the time. In the case below:

I have studied maths. Do you accept this as an A-level subject? I would also like to enquire as to whether a computing foundation year course from another university would be accepted as a means of entry to the Computer Science with

Security and Forensics course?

The last two sentences would be identified as questions and all topics would be identified as "Qualifications". The first sentence would be combined with each question and searched for most similar. Of course this is wrong, only the first question should be attached with the sentence. A way to work around this would be to make the labels more specific, but currently with amount of data I had this would not be feasible.

6 Future Work

6.1 Check responses to store

Starting with the initial use of the system, unless the user already has data built up that they can use, the data stored will be minimal, which will cause the system to initially make mistakes. These mistakes mentioned above will lead to a build up of bad data in the case base which will cause the system to never function correctly. Therefore it is vital that something is put in place to stop this from happening. My way to solve this would be to add a checkbox for each sentence the system has decided is a question and let the user check what responses they wish to store in the case base.

6.2 Spelling correction

As with any system we should be anticipating that the user will make some mistakes. Whether it be the original message sender or the current user of the system. In our case, as we are working with text, trying to classify, parse and identify similar text blocks, mistakes here can have a large impact on the overall quality of the end result. The key mistake and most common mistake I am referring to is spelling mistakes. As the vectors used in most features of the system are created using the words in the sentence, a mistake in these vectors will impact the similarity between two vectors and could have an impact on the classification given.

Say we have 20 questions in total including the one below and the one to compare

What are the minimum qualifications that need to be met?

and we have two examples to compare one with a spelling mistake and one without.

Are there any minimum qualifications that I will need to meet?

Are there any minnum qualifications that I will need to meet?

If we convert them to possible tf-idf vectors we could have:

1 [0.022, 0, 0, 0.23, 0.19, 0.016, 0, 0, 0.07, 0.01, 0, 0.06, 0.022, 0.105, 0.161, 0]

2 [0.02, 0.015, 0.073, 0.209, 0.172, 0.015, 0.005, 0.039, 0.063, 0.01, 0.146, 0, 0, 0, 0, 0]

3 [0.02, 0.015, 0.073, 0, 0.172, 0.015, 0.005, 0.039, 0.063, 0.01, 0.146, 0, 0, 0, 0, 0.272]

The cosine similarity between 1 and 2 is 0.716.

The cosine similarity between 1 and 3 is 0.278.

So we can see here what a different one spelling mistake can make, this will be the difference between a correct response being selected or not. The way to minimise this would be to implement some kind of spelling correction that could handle mistakes when loading in emails and when free typing.

6.3 Feedback to user

As the amount of training data increases the amount of time to train the model increases. On selecting the button to train there is no current feedback to the user to let them know that training is taking place or in what stage of the process it is at. All users will want to know the state of the system so they know when they can carry on using it. As the amount of cases stored also increases the time to generate the response will also increase as it will have more questions to measure similarity against. Both of these stages should provide feedback to the user. This could be a waiting wheel at minimum, but preference would be a progress bar, as it gives more indicating on the amount of time remaining.

6.4 Polish UI

The look and feel of the system is very import as it is what the user sees and interacts with. A poor user interface can make or break a system and can even go as far as stop people from using it no matter how good the functionality. To improve what I currently have I believe the email list view could be improve to replicate more of what we see in an email client. I do not believe any of the current widgets in PyQt can give us this look so a custom widget will have to be created. Things start to get a bit messy when unanswered questions are rendered in the window, particularly when they are longer. It seems other widgets are made smaller to make room in the window. Some sort of fixed sizing here would be preferable.

6.5 Support for more emails

In most cases more options for the user is always preferable. In the cases of emails there is a vast amount of options in terms of accounts. With people

having personal and work emails often from different providers it would be nice for them to be able to use any account within the system. Currently the only supported account is Outlook Office 365 as it was easy to test through my university email account, but obviously popular options such as gmail should be added. Google actually has a gmail API available that would be used to implement this feature. Options for other accounts could allow users to input IMAP and SMTP server addresses and ports.

6.6 Paragraph vector

As stated one of the most important features is the find the most similar question in the case base. The problem with the bag-of-words tf-idf model is the ordering of the words is lost and the semantics of the words are ignored. Trying different approaches here would be nice to see if we can produce better results. Google have proposed an unsupervised learning algorithm here [13] called Paragraph Vector. The way it works is using prediction of the next word to appear in the paragraph given the context. This is achieved by using a combination of a matrix of trained paragraph vectors and word vectors in the current context. Observed results have shown that it outperforms many other techniques for representing text. On text classification such as what I am using it has given improvement of around 30%. Using this approach improves on the two issues mentioned above that the bag-of-words model has.

7 Conclusion

Email is one of the key forms of communication today and an automated tool would put us in a position to save time and increase productivity. To make strides towards this I have proposed a system that utilises natural language processing and machine learning techniques that continually improves as it receives more emails to generate responses from.

The main aim for the project was to try and add some level of automation to email response, to go about achieving this I have focused on a case based reasoning approach where previous emails and responses are stored and used to respond to new messages. My area of focus was the role of an admissions tutor due to the availability of data that I could have access to.

To achieve this I have had to address a few challenges:

- The identification of questions which was established by parsing each sentence and looking for SBARQ and SQ tags if this could not be found and their exists an SBAR tag then a classifier is used to determine at question or not.
- The adaption of emails is a difficult task and not a well researched area. I have used a different approach where I am splitting down emails and

using composition to build a response from a list of previous responses.

- The approach above introduced a new problem of the different styles of writing people have and relating important information to each question. This was solved by classifying each sentence and combining matching topics.
- The key aspect is the way to measure how similar documents are to one another. To do this I am using latent semantic analysis to create vectors from the sentence and using the cosine similarity to obtain a similarity value.

To test the system I have made use of a combination of automated unit testing and manual testing. The unit tests can be run to make sure the inner workings of the system are functioning correctly and the manual tests handle the user interface and functionality behind loading and sending emails - common work flows that a user would easily notice problems with.

Although all requirements have been met the quality of the system is hard to analyse from a short testing phase. Even though the system does make mistakes at this stage in the identification of questions and topics, this is likely due to the small amount of training data the system has used.

I have proposed a list of future enhancements to the system which both aid in the fixing of some of the key issues identified, improving the usability and performance. This includes:

- Adding checkboxes to allow the user to select what responses to store.
- Adding spelling correction to stop user error effecting similarity comparison.
- Providing feedback to the user to improve usability.
- Updating the UI to improve the familiarity and usability.
- Adding support for more emails to provide more flexibility to users.
- Implementing paragraph vectors to check for improved vector construction

Overall the system provides some level of automation to email response but initially will require a lot of input from the user. Provided the email is the first in the conversation it is possible for a reasonable response to be given. To work for all cases a way for the system to remember the context of the conversation would need to be implemented.

8 Reflection

In this stage of my career most projects will pose a range of difficulties that I will have not encountered before and present challenges and learning that takes time. Most of the assumptions made at the start of the project had in fact changed drastically throughout which made a big impact on the way the system worked.

From the start I had assumed I would be developing the system using Java, as this is what I had most experience with using it for most programming modules throughout the course. I had knowledge that there were machine learning and natural language processing libraries that could help with some of my tasks. Most of these libraries exist for Python but not Java. So I had the decision of using Java and writing all of the code myself or using Python. I have very little experience using Python, only a module in first year, so initial time to brush up on my skills would be required. I decided to use Python as I believed it to be unnecessary to re implement features just because of a language preference, so deemed Python more suitable. This gave me an appreciation of selecting an appropriate tool to perform a task and not just sticking to what I am familiar and comfortable with. Being out of our comfort zone is a good way to enhance learning [14] and widen our knowledge areas. Hence giving me more options in my ?comfort zone? for later projects.

As my approach evolved the identification of questions became key to the inner workings of the system. As an email has been sent I am assuming that a question will always be in there, otherwise there would be no need to send the email in the first place. The different approaches mentioned above in challenges outlines the different methods attempted to provide the most accurate discovery of questions. Each step taken has removed an assumption about how questions will be written until we get to the stage where there are no assumptions in place. This is a way of looking at and sort of breaking down a problem that I had never used before and gives me a new way to solve problems that I never thought of using before. This task is difficult to achieve and still the method used does not 100% guarantee that just questions will be identified. For the time constraints on the project sometime compromises have to be made and I believe this is the best I could have got it. Problems like this helps you understand that sometimes perfection is not achievable due to restriction that are in place or to the difficulty of the problem, but getting as close as possible is what we can hope to achieve. This has also shown me another area to which classification can be applied.

A less substantial part of the project is the UI and took up more time than perhaps it should. This time could have been spent elsewhere improving the quality of the email response itself. Usability is a key factor here but what the UI was written in is not. Possibly creating a web application instead may have been a better decision as I have more experience in this area. In reality there

was no thinking behind the approach I took which was a mistake on my part. The case of using Python is a bit different to what we have here as Python was selected due to the libraries available and the functionality used there was crucial to the workings of the system. In the case of the UI selecting something that I could work with quickly would have been the better choice. In the future this will help me thinking about my options before getting straight to coding. It will also help me make better decisions by analysing the current situation including time constraints and important parts of a project.

References

- [1] I. G. P. A?airs. *Interconnected World: Communication and Social Networking*. URL: <http://www.ipsos-na.com/news-polls/pressrelease.aspx?id=5564>.
- [2] Anjuli Kannan et al. “Smart Reply: Automated Response Suggestion for Email”. In: (2005).
- [3] Ann Bies et al. “Bracketing Guidelines for Treebank I I Style Penn Treebank Project”. In: (1995).
- [4] Hinrich Schutze Christopher D. Manning Prabhakar Raghavan. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [5] *Cosine Similarity*. URL: https://en.wikipedia.org/wiki/Cosine%5C_similarity.
- [6] scikit-learn developers. *Model persistence*. URL: http://scikit-learn.org/stable/modules/model%5C_persistence.html.
- [7] scikit-learn developers. *sklearn.feature_extraction.text.TfidfVectorizer*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.feature%5C_extraction.text.TfidfVectorizer.html.
- [8] scikit-learn developers. *sklearn.linear_model.SGDClassifier*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.linear%5C_model.SGDClassifier.html#sklearn.linear%5C_model.SGDClassifier.
- [9] The Python Software Foundation. *imaplib IMAP4 protocol client*. URL: <https://docs.python.org/3/library/imaplib.html>.
- [10] The Python Software Foundation. *smtplib SMTP protocol client*. URL: <https://docs.python.org/3/library/smtplib.html>.
- [11] Stanford CoreNLP Group. *CoreNLP Server*. URL: <https://stanfordnlp.github.io/CoreNLP/corenlp-server.html%5C#api-documentation>.
- [12] Luc Lamontagne and Guy Lapalme. “APPLYING CASE-BASED REASONING TO EMAIL RESPONSE”. In: ().
- [13] Quoc Le and Tomas Mikolov. “Distributed Representations of Sentences and Documents”. In: ().

- [14] Andy Molinsky. *If You're Not Outside Your Comfort Zone, You Won't Learn Anything*. URL: <https://hbr.org/2016/07/if-youre-not-outside-your-comfort-zone-you-wont-learn-anything>.
- [15] Ibrahim Naji. *10 Tips to Improve your Text Classification Algorithm Accuracy and Performance*. URL: <http://thinknook.com/10-ways-to-improve-your-classification-algorithm-performance-2013-01-21/>.
- [16] Maja Pantic. *Introduction to Machine Learning & Case-Based Reasoning*. URL: <https://ibug.doc.ic.ac.uk/media/uploads/documents/courses/syllabus-CBR.pdf>.
- [17] *Tf-Idf*. URL: <http://www.tfidf.com>.
- [18] Alex Thomo. "Latent Semantic Analysis". In: ().
- [19] Jana Vembunarayanan. *Tf-Idf and Cosine similarity*. URL: <https://janav.wordpress.com/2013/10/27/tf-idf-and-cosine-similarity/>.
- [20] Rosina O Weber and Kevin Ashley. "Textual Case-Based Reasoning". In: (2005).