A

REPORT

ON

Developing a Search Engine on a nuclear corpus and outlining a semantic based approach to Entity Profiling from raw text to build a Question Answering system

BY

Skanda Vaidyanath **2016A7PS0236H** **Computer Science**

Practice School-1 Course No.

BITS F211

At

Indira Gandhi Centre for Atomic Research

A Practice School-1 station of

Birla Institute of Technology and Science, Pilani

May 22nd- July 13th 2018

A

REPORT

ON

Developing a Search Engine on a nuclear corpus and outlining a semantic based approach to Entity Profiling from raw text to build a Question Answering system

BY

Skanda Vaidyanath **2016A7PS0236H**

At

Indira Gandhi Centre for Atomic Research

Birla Institute of Technology and Science, Pilani

May 22nd-July 13th 2018

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)

Practice School Division

Station: Indira Gandhi Centre for Atomic Research **Centre:** Kalpakkam

Duration: From 22nd May 2018, **To: 13th July 2018**

Date of Submission:

Title of the Project: **Developing a Search Engine on a nuclear corpus and outlining a semantic based approach to Entity Profiling from raw text to build a Question Answering system**

ID No. 2016A7PS0236H

Name: Skanda Vaidyanath

Discipline: Computer Science Engineering

Name of the Guide: Mr. Subba Raju

Designation of the Guide: Technical Officer (D)

Key Terms: Information Retrieval, Document Ranking, Sentence Extraction, Recommender System, Neural Networks, Stacked Autoencoders, Natural Language Processing, Entity Profiling, Attribute Extraction, Question Answering

1. ABSTRACT

Entity Profiling is the task of identifying entities of interest in structured/ unstructured text and extracting information about them. We decide on attributes of these entities that we wish to extract and use various Information Extraction techniques to identify values for these attributes. There are many approaches to the task of Entity Profiling, many involving Supervised and Semi-supervised learning techniques. In this paper, we focus on a semantic approach to Entity Profiling, which has many benefits over the other methods. The end goal of this project is to use this technique to devise a Question-Answering system for factoid-based questions. This is a novel approach to solving the problem of Question-Answering and has not been solved extensively before this. As an introduction to the task of Question-Answering, we solve the simpler problem of Document Ranking and Sentence extraction using conventional Information Retrieval techniques. Therefore the first part of this paper discusses the details of this statement and its shortcomings, thus giving us a clear reason as to why we had to move on to newer, state of the art techniques. However, we will only outline the approach that we will follow in the Entity Profiling step and not give an implementation as that is beyond the scope of this report. But, we will delve rather deep into the implementation of the search engine that we create, including some features like a spell-correction feature, an autocomplete feature and even a feature that recommends appropriate search results to the user.

Signature of the student:

Date:

Signature of the PS faculty:

Date:

2. ACKNOWLEDGEMENTS

I would like to thank **Dr. A.K. Bhaduri**, Director, IGCAR and also BITS - Pilani for giving me an opportunity to undergo Practice School - 1 training at IGCAR, a renowned research institute in India.
I would like to thank **Dr. B. Venkataraman**, Director, RMPAG for his kind support and motivation for this practice school programme.
I would like to express my sincere gratitude to **Dr. T.S. Lakshmi Narasimhan**, Associate Director, Resources Management Group, IGCAR, practice school - 1 coordinator and also **Dr. Vidya Sundararajan**, Head, Planning and Human Resource Management Division for their immense help and valuable guidance in conducting this programme.
My sincere thanks to **Dr. V.S. Srinivasan**, Head, LIMS, SIRD for his kind assistance, cooperation and constant encouragement in various practice school programme related activities.
I am thankful to my project supervisor **Mr. Subba Raju**, Technical Officer (D), Computer Division, Electronics and Instrumentation Group, IGCAR for his constructive guidance in carrying out the project.
I am really grateful to **Dr. Harihara Venkataraman**, PS-1 Faculty, BITS - Pilani, Hyderabad Campus, for his kind cooperation and guidance during my entire stay at Kalpakkam.
I would also like to thank **Mr. R. Srikanthan** and **Mr. R. Ramesh Babu** for their kind coordination in practice school - 1 related activities.

3. TABLE OF CONTENTS

Contents

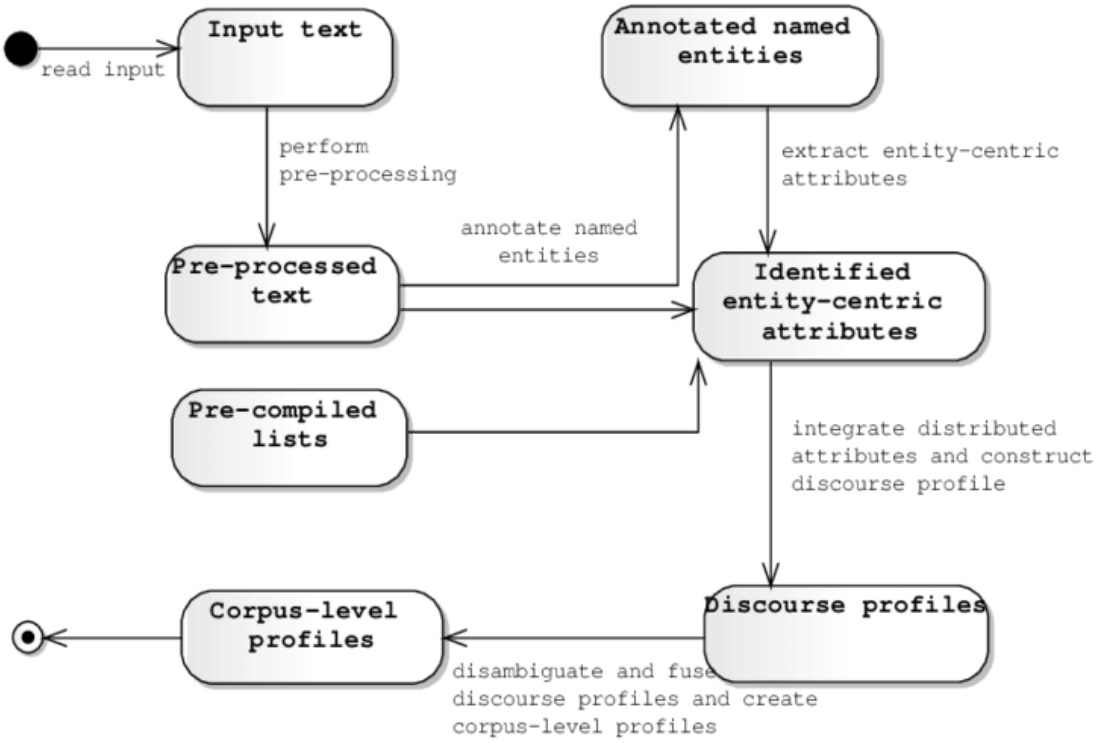
1. INTRODUCTION

Entity Profiling is the task of identifying entities of interest in structured/ unstructured text and extracting information about them. It is often seen that in many applications, we would like to extract information about a particular entity from a huge amount of structured/unstructured text data. An entity could be anything to which a proper noun maybe associated – Person, Location, Organization, Product, etc. This information that we wish to extract could be in the form of attribute-value pairs. This means that the onus is on the developer to come up with appropriate attributes for each entity type. For example, for a Person entity, some appropriate attributes could be Full Name, Occupation, Place of Work / Organization of affiliation, Date of Birth, Place of Birth and so on. The task is to extract the values of these attributes from text data and create a database of information for each named entity. A quick word on structured and unstructured data wouldn't be remiss here. Crudely, structured data is one that is in the form of short phrases that probably already directly contains the information we need. It does not follow standard writing norms. Unstructured data on the other hand, is how general text is written. It follows standard writing norms and we need specialized Information Extraction techniques to extract the attribute values from this data. There are several approaches to solve the problem of Entity Profiling, many of which are quite recent developments. These techniques use Supervised and Semi-supervised learning techniques, which we will not get into at this stage. Our approach is a more semantic based approach that offers some significant benefits over the other approaches. It is based on a number of concepts in Natural Language processing which allows the method to exploit the semantic information in the text. It allows for syntactic and structural variation and ensures extraction of attribute values at the cross-document and cross-sentence level. Having created a database of these named entities, the idea is to use this database for a Question-Answering system, especially tailor made for factoid-based questions. The assumption is that, after the Entity Profiling step, the answers/ facts we seek about said named-entity will be readily available in the database. Question-Answering is one of the most challenging tasks in Natural Language Processing and people have tried to solve the problem in a number of different ways. Some of these techniques use Natural Language processing, some use Neural Networks but not many attempt to pre-create an entity profile to use to solve this problem. Our project is an attempt to see if we can make any progress in this area and check if Entity Profiling could be a viable and accurate solution to the Question-Answering problem. Since this is an extremely complex problem, we have not attempted to give an implementation of this approach. We only give an outline of the steps to be followed.

Before we embark on the challenging task of Question-Answering, we devise a simple search engine that accepts a query and ranks the relevant documents (Document Ranking) in order and also picks out one main sentence (Sentence Extraction) from the document corpus for the user to review straight away. This technique uses conventional Information Retrieval techniques like the Vector Space Model and Word Embeddings to solve this problem. We will give a detailed outline of this system before we move on to the problem of Entity Profiling. However, this technique has its own limitations, which we will talk about as well. Section 1 of this paper will discuss the Document Ranking and Sentence Extraction model. We also talk about some additional features in this search engine like a spell-correction feature, an auto-complete feature and even a recommendations feature that recommends the most appropriate search results for the user. Here we will talk about the exact details of these broad steps and even give some code snippets. At the end of this section, we discuss the limitations of this model. In Section 2 we discuss the novel technique of Entity Profiling and how it can

be used for Question-Answering. In the final section (Section 3), we make conclusions and summarize the work we have done and talk about future work.

xplain the components of our proposed method in more details.



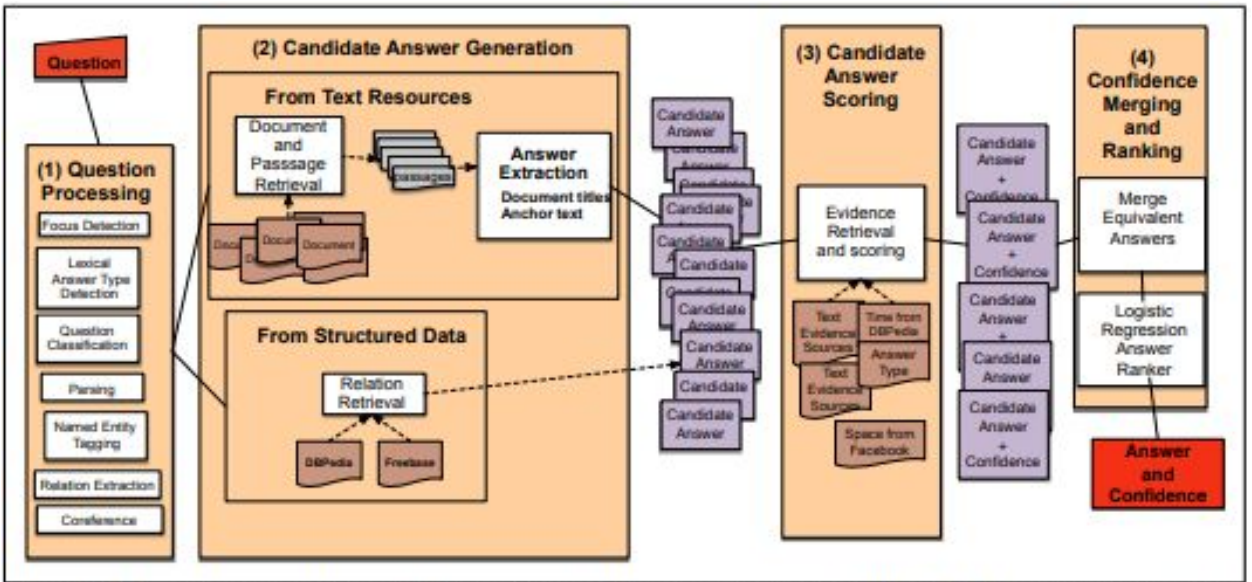


Figure 28.9 The 4 broad stages of Watson QA: (1) Question Processing, (2) Candidate Answer Generation, (3) Candidate Answer Scoring, and (4) Answer Merging and Confidence Scoring.

2. A SIMPLE SEARCH ENGINE

We divide this section into two parts – the Document Ranking step and the Sentence Extraction step. Note that we split the entire code into "setup" code and "live" code. The setup code is run only once and sets up the system and the live code is run over and over again to process queries.

Document Ranking

This step consists of seven subparts.

Pre-processing

Given a corpus of documents, that may be present in a directory-tree like structure, the initial task is to access these documents one by one and write them all in one single file, such that each document occupies one line. This will be useful for further processing. We create one single text file with the entire corpus in it, each line containing one document. After this, we apply some simple preprocessing steps to each document. We start by tokenizing the document, then removing the stop-words and punctuations and numbers and finally we stem all the words as well. This level of pre-processing is sufficient for our task, as we learnt by reading through various past examples and works. At the end of this step, we get a list of lists, where each inner list represents the document in tokenized, preprocessed form. The outer list represents the entire corpus. Also note that we are streaming in the documents from the file, one after the other here, so that we don't hold the entire corpus in main memory at any point in time. We save this preprocessed list of lists as a serialized file to use later. Since this part doesn't change, we don't need to run this code every time. Therefore this would fall under the setup part of the code.

```
path = "./test3"
file3 = open("cmptext.txt","w+")
number_of_documents = recursive_read(path,file3)
file3.close()
print 'All files read'
file3 = open("cmptext.txt","r")
preprocess(file3, number_of_documents)
file3.close()
print 'All files processed'
```

```
def read_pdf(filename):
    pdfFileObj = open(filename, 'rb')
    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
    num_pages = pdfReader.numPages
    count = 0
    text = ""
    scanned = False
    try:
        while count < num_pages:
            pageObj = pdfReader.getPage(count)
            count += 1
            text += pageObj.extractText()
        if text != "":
            text = text
        else:
            text = textract.process(filename, method = 'tesseract', language = 'eng')
            scanned = True
    except Exception as e:
        raise e
    return text, scanned
```

```
def recursive_read(path,file3):
    number_of_documents = 0
    for filename in os.listdir(path):
            filenm = path + "/" + filename
            if(os.path.isdir(filenm)):
            number_of_documents += recursive_read(filenm,file3)
        else:
            try:
                text, scanned = read_pdf(filenm)
                if not scanned:
                    text = text.encode('utf-8').split('\n')
                else:
                    text = text.split('\n')
                for word in text:
                    file3.write(word)
                    file3.write(' ')
                del text[:]
                file3.write('\n')
                print 'Document Read'
                number_of_documents += 1
            except Exception:
                print scanned
                print '!!!!!!!!!  WARNING READ FAILED !!!!!!!!'
    return number_of_documents
```

```
def preprocess(file_name, number_of_documents):
    stemmer = PorterStemmer()
    fp1 = open("preprocessed.txt", "wb")
    fp2 = open("preprocessed-cmptext.txt", "wb")
    pickle.dump(number_of_documents, fp1)
    for line in file_name:
        preprocess_list1 = gensim.utils.simple_preprocess(line, max_len = 20)
        preprocess_list2 = []
        for word in preprocess_list1:
            if word not in stop_words:
                preprocess_list2.append(word)
        pickle.dump(stemmer.stem_documents(preprocess_list2), fp1)
        for word in preprocess_list2:
            fp2.write(stemmer.stem(word.encode('utf-8')))
            fp2.write(' ')
        fp2.write('\n')
    fp1.close()
    fp2.close()
```

Word2Vec and Vocabulary generation

Word2Vec is a Word Embedding technique developed by Google. It is based on Neural Networks and it converts words to vectors and plots them in a vector space. It is based on one of two different models – the Skip Gram Model or the Continuous Bag of Words. They do not give significantly different results and we have used the Skip Gram Model for our application. The Neural Networks are able to learn the features of the word so well that it plots the words in the vector space such that similar words appear close to each other and dissimilar words appear farther away. In fact, the model is so powerful that relations like the following hold:

Vec('King') – Vec('Queen') + Vec('Man') = Vec('Woman')

This is an extremely non-technical explanation of the model as it does not play an extremely important role in our particular application. The in-depth analysis of the neural networks is quite rigorous and not straight-forward. Having said that, it is not required here.

In our application, we will use Word2Vec for Query Expansion (Refer Section 1.1.5). But the application of this model also gives us the vocabulary of all words in our corpus. We have ensured that this vocabulary also includes important bigrams (two word phrases) and trigrams (three word phrases) for better accuracy. We also ensure that the documents are streamed in from our preprocessed file so that at no point in time are all the documents in the main memory at the same time. This is part of the setup code as well and we save away this Word2Vec model for use later.

```
print 'Word2Vec begins'
model = get_word2vec(number_of_documents)        #includes trigrams
model.save('vocab.txt')
print 'Word2Vec done'
vocabulary = model.wv.vocab.keys()
```

```
def get_word2vec(number_of_documents):
    documents = gensim.models.word2vec.LineSentence("preprocessed-cmptext.txt")
    bigram = gensim.models.Phrases(documents)
    trigram = gensim.models.Phrases(bigram[documents])
    model = gensim.models.Word2Vec(trigram[bigram[documents]],size=300,window=10,min_co
    model.train(documents, total_examples=number_of_documents, epochs=10)
    return model
```

TF-IDF Vectors for all Documents

This is a very important step and our entire algorithm for document ranking hinges on this step. Before we proceed any further, an explanation of the Vector Space Model of Document ranking is warranted. It is an extremely popular model, used extensively in Information Retrieval. We will also talk about TF-IDF scoring a little bit to give the readers some clarity.

The Vector space model is an Information Retrieval paradigm that plots documents of a corpus and the query as a vector in a vector space and tries to find the document most similar to the query. The vector space is |V| dimensional where |V| is the number of words in our vocabulary we found in the previous step. Each document and the query are represented as |V| dimensional vectors where each value in the vector corresponds to one word in the vocabulary. This entry corresponding to each word is based on some scoring and the one we will use in this application is the TF-IDF scoring technique. At the end of this process we must have a document vector for each document and one vector for the query as well. Similarity is calculated based on Cosine Similarity. This is basically the cos(theta) term in conventional dot product of vectors. The smaller the cosine similarity, lesser is the similarity between the document and the query and vice versa. The idea is to pick the ‘n’ most similar documents.

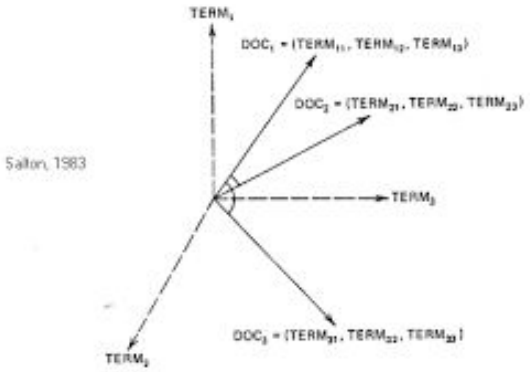


Figure 4-2 Vector representation of document space.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

The only question left unanswered here is – What are the values we enter in the document vector or the query vector? This is where TF-IDF scoring comes in. TF-IDF stands for Term Frequency – Inverse Document Frequency. The formula has two parts and the final score is the product of the TF score and the IDF score. Term frequency gives a measure of the number of times a particular word w appears in a given document d. Inverse document frequency on the other hand gives an idea of how common that particular word is in the entire corpus. This ensures that words that are not exactly specific to a particular document, but common to all documents are not scored very high because of their large term frequencies. There are multiple formulas for both TF and IDF as shown below and we can use them in any combination we like. In this project, we use the log normalization formula for TF and the inverse document frequency formula (from the image below) for IDF. Note that this is a Bag of Words model that means that the order of the words itself do not carry any meaning. Only the frequencies of each word carry a meaning. This could be interpreted as a drawback of the model.

Variants of term frequency (TF) weight

weighting scheme	TF weight
binary	$0, 1$
raw count	$f_{t,d}$
term frequency	$f_{t,d} \Big/ \sum_{t' \in d} f_{t',d}$
log normalization	$\log(1 + f_{t,d})$
double normalization 0.5	$0.5 + 0.5 \cdot \frac{f_{t,d}}{\max_{\{t' \in d\}} f_{t',d}}$
double normalization K	$K + (1 - K)\frac{f_{t,d}}{\max_{\{t' \in d\}} f_{t',d}}$

Variants of inverse document frequency (IDF) weight

weighting scheme	IDF weight ($n_t = \lvert\{d \in D : t \in d\}\rvert$)
unary	1
inverse document frequency	$\log \frac{N}{n_t} = -\log \frac{n_t}{N}$
inverse document frequency smooth	$\log\left(1 + \frac{N}{n_t}\right)$
inverse document frequency max	$\log\left(\frac{\max_{\{t' \in d\}} n_{t'}}{1 + n_t}\right)$
probabilistic inverse document frequency	$\log \frac{N - n_t}{n_t}$

Now coming back to our project, the idea is to pre-calculate the TF-IDF vectors of all the documents and store them away since they do not change. This saves computational time when we run the live code. Now that we know how to calculate the TF-IDF score given a word and a document, we can find the TF-IDF vectors for all documents. Note that we have used the logarithmic formulae for both TF and IDF. We also pre-calculate the norms of the vectors we just found. This is essentially the magnitude of the vector which we will use when determining similarity scores. Also note that the TF-IDF calculation has been “streamed” as well i.e. we do not load the entire data back into memory. Rather we load the data, one document at a time.

```
def get_tfidf_vectors(inverted_index, length_preprocessed):
    document_tfidf = dict()
    fp1 = open("tfidf-scores.txt", "wb")
    pickle.dump(length_preprocessed, fp1)
    fp2 = open("preprocessed.txt", "rb")
    length_preprocessed = pickle.load(fp2)
    for _ in range(length_preprocessed):
        document = pickle.load(fp2)
        for word in inverted_index.keys():
            value = calc_wt(document, word, inverted_index, length_preprocessed)
            document_tfidf[word] = value
            pickle.dump(document_tfidf, fp1)
    fp1.close()
    fp2.close()
```

```
def calc_wt(text, word, inverted_index, length_preprocessed):
    if "_" in word:
        if len(text) > 30:
            n_words = word.split("_")
            number_of_occurences = 0
            if (len(n_words) == 2):
                for i in range(len(text)-1):
                    if text[i] == n_words[0] and text[i+1] == n_words[1]:
                        number_of_occurences += 1
            if (len(n_words) == 3):
                for i in range(len(text)-2):
                    if text[i] == n_words[0] and text[i+1] == n_words[1] and text[i+2] == n_words[2]:
                        number_of_occurences += 1
            if (len(n_words) == 4):
                for i in range(len(text)-3):
                    if text[i] == n_words[0] and text[i+1] == n_words[1] and text[i+2] == n_words[2] and text[i+3] == n_words[3]:
                        number_of_occurences += 1

            tf = log(1 + number_of_occurences,10)
        else:
            tf = log(1 + text.count(word),10)
        idf = log((float(length_preprocessed)/len(inverted_index[word])),10)

    else:
        tf = log((1 + text.count(word)),10)
        idf = log((float(length_preprocessed)/len(inverted_index[word])),10)
    return tf*idf
```

```
def get_norms():
    fp1 = open("norms.txt", "wb")
    fp2 = open("tfidf-scores.txt", "rb")
    nod = pickle.load(fp2)
    dic_of_norms = dict()
    doc_number = 1
    for _ in range(nod):
        norm = 0
        dic = pickle.load(fp2)
        for item in dic.values():
            norm += item**2
        norm = norm**0.5
        dic_of_norms[doc_number] = norm
        doc_number += 1
    pickle.dump(dic_of_norms, fp1)
    fp1.close()
    fp2.close()


get_tfidf_vectors(inverted_index, number_of_documents)
get_norms()
```

At the end of this step, we have pre-calculated document vectors for each document and the norms of these vectors as well. Needless to say, this comes under the setup stage of the code.

Building an Inverted Index

(This step is done before the TF-IDF scoring in the implementation)

An inverted index is a data structure where we store, for each word in our vocabulary, the documents in which that word is present. For example, say the word ‘sodium’ is present in document numbers 1,4,12, 45, 67 and 89. We have a Python dictionary with the keys as all the words from the vocabulary and for each key, the associated value is a list with all the document numbers. Therefore, each key : value pair in the dictionary looks like - ‘sodium’ : [1, 4, 12, 45, 67, 89]. We do this for bigrams and trigrams as well.

To build this data structure, we iterate over all the words in the vocabulary and find all the documents that have this particular word using the pre-processed list of lists that we already created. Since this is a computationally expensive task, we do this in the setup stage and store our results since this index structure doesn’t change with time.

```
inverted_index = get_inverted_index(vocabulary)
for item in inverted_index.keys():
    if not inverted_index[item]:
        del inverted_index[item]
with open("inverted-index.txt", "wb") as fp:
        pickle.dump(inverted_index, fp)
fp.close()
```

```
def get_inverted_index(dictionary):
    inverted_index = dict((el,[]) for el in dictionary)
    fp = open("preprocessed.txt", "rb")
    length_preprocessed = pickle.load(fp)
    doc_number = 1
    for _ in range(length_preprocessed):
        document = pickle.load(fp)
        for el in inverted_index.keys():
            if el in document:
                inverted_index[el].append(doc_number)
            elif '_' in el:
                n_words = el.split('_')
                result = False
                if (len(n_words) == 2):
                    for i in range(len(document)-1):
                        if result:
                            break
                        if document[i] == n_words[0] and document[i+1] == n_words[1]:
                            result = True
                if (len(n_words) == 3):
                    for i in range(len(document)-2):
                        if result:
                            break
                        if document[i] == n_words[0] and document[i+1] == n_words[1] and document[i+2] == n_words[2]:
                            result = True
                if (len(n_words) == 4):
                    for i in range(len(document)-3):
                        if result:
                            break
                        if document[i] == n_words[0] and document[i+1] == n_words[1] and document[i+2] == n_words[2] and document[i+3] == n_words[3]:
                            result = True
```

```
                if result:
                    inverted_index[el].append(doc_number)
            else:
                continue
        doc_number += 1
    fp.close()
    return inverted_index
```

Query Expansion and getting the query vector

This step is part of the live part of the code since it involves computation involving the query. Once the user has entered the query, we want to “expand” the query to accommodate some terms that might not already be a part of the query. We also try to identify phrases in the query, which might be crucial in getting relevant and accurate search results. In fact this entire step plays a huge role in determining the accuracy of our system. Also at the end of this step, we will calculate the query vector in the same way that we calculated the vectors for all the documents. We also calculate the query norms.

We perform a couple of steps during the query expansion step. First, we pre-process the query in the very same way that we pre-processed the documents. Next, we try to identify phrases in the expanded query. We process the query from left to right and try to identify bigrams and trigrams. If we find a bigram or a trigram, we remove the corresponding unigrams that form that bigram/ trigram. This is done to improve the accuracy of our results. Further, based on empirical results, we may remove the bigram constituents of the trigrams as well, if it is found that it produces better results. Another key step is expanding the query using the Word2Vec that we created. It is difficult to expect the user to enter the exact terms that are found in the corpus in their query. So we expand the query by adding some words that are closely related to the query words. Empirically, for each query word, we add three other similar words from our Word2Vec model. This is a good approximation for large corpuses. Finally, we take only

the set (mathematical set) of the words i.e. unique words that are present in our expanded query.

The next task is to find the TF-IDF score vector for this expanded query. The task is very similar to the document vector finding task, with the only difference being that in this case, we consider the expanded query itself as a document. We also calculate the norm of this query vector for further similarity calculations.

```
def get_expanded_query(query, model, vocab):
    expanded_query = []
    for word in query:
        expanded_query.append(word)
        '''for phrase in vocab:
            if '_' in phrase:
                n_words = phrase.split('_')
                if word in n_words:
                    expanded_query.append(phrase)'''
    unnecessary = set()
    for i in range(len(query)-1):
        if (str(query[i]) + '_' + str(query[i+1])) in vocab:
            expanded_query.append(str(query[i]) + '_' + str(query[i+1]))
            unnecessary.add(query[i])
            unnecessary.add(query[i+1])

    for i in range(len(query)-2):
        if (str(query[i]) + '_' + str(query[i+1]) + '_' + str(query[i+2]) ) in vocab:
            expanded_query.append(str(query[i]) + '_' + str(query[i+1]) + '_' + str(query[i+2]))
            unnecessary.add(query[i])
            unnecessary.add(query[i+1])
            unnecessary.add(query[i+2])

    for i in range(len(query)-3):
        if (str(query[i]) + '_' + str(query[i+1]) + '_' + str(query[i+2]) + '_' + str(query[i+3])) in vocab:
            expanded_query.append(str(query[i]) + '_' + str(query[i+1]) + '_' + str(query[i+2]) + '_' + str(query[i+3]))
            unnecessary.add(query[i])
            unnecessary.add(query[i+1])
            unnecessary.add(query[i+2])
            unnecessary.add(query[i+3])

    number_of_similar_words = 0
    for token in query:
        a = []
        if token not in vocab:
            continue
        a.extend([x[0] for x in model.wv.most_similar(token)[0:(number_of_similar_words)]])
        expanded_query.extend(a)
    expanded_query = set(expanded_query)
    for word in unnecessary:
        expanded_query.remove(word)
    return expanded_query
```

```
def get_eq_tfidf_vector(inverted_index, expanded_query, length_preprocessed):
    eq_tfidf = dict()
    expanded_query = list(expanded_query)
    for word in inverted_index.keys():
        value = calc_wt(expanded_query, word, inverted_index, length_preprocessed)
        eq_tfidf[word] = value
    return eq_tfidf
```

Getting the Relevant documents

We have one final step before we get down to the final document ranking step. It is an extremely intensive task (computationally) to find the similarity scores with respect to the query for all the documents in the corpus. Therefore, we use the inverted index that we created to create a set of relevant documents to find similarities. These relevant documents are all those

documents that have at least one word from the expanded query in them. We go through every word in the expanded query and find all the relevant documents for each word and take a union to get the final, entire set of relevant documents. We are now ready to do the final document ranking and extract the top ten relevant documents.

```
def get_relevantdocs(expanded_query, inverted_index):
    relevantdocs = set()
    for token in expanded_query:
        if(token not in inverted_index.keys()):
            continue
        relevantdocs.update(inverted_index[token])
    return relevantdocs
```

Similarity scores and Final Ranking

We are now ready to do the final document ranking on the set of relevant documents. For each document in the set of relevant documents, we calculate the Cosine Similarity as described in Section 1.1.3. The document with the highest Cosine Similarity score is the one that is ranked the highest. And in this manner, the top ten documents are the ones with maximum Cosine Similarity scores. We use a max heap to store all the relevant documents and their scores and extract the top ten with a simple heap-pop operation. We now have a final list of the top ten ranked documents.

```
def get_scores(relevant_docs, eq_vector):
    fp1 = open("tfidf-scores.txt", "rb")
    fp2 = open("norms.txt", "rb")
    norm_dic = pickle.load(fp2)
    query_norm = 0
    for item in eq_vector.values():
        query_norm += item**2
    query_norm = query_norm**0.5
    nod = pickle.load(fp1)
    doc_number = 1
    scores = dict((el,0) for el in relevant_docs)
    try:
        while doc_number <= nod:
            doc_vector = pickle.load(fp1)
            if doc_number in relevant_docs:
                for word in eq_vector.keys():
                    scores[doc_number] += eq_vector[word]*doc_vector[word]
                scores[doc_number] /= ((query_norm)*(norm_dic[doc_number]))
            doc_number += 1
    except ZeroDivisionError as e:
        raise e
    fp1.close()
    fp2.close()
    return scores
```

```
try:
    scores = get_scores(relevant_docs, eq_vector)
    if not scores:
        print 'Match Not Found.'
        continue
except ZeroDivisionError:
    print 'Please be more specific.'
    continue
heap_docs = [(-value, key) for key,value in scores.items()]
largest_docs = heapq.nsmallest(10, heap_docs)
largest_docs = [(key, -value) for value, key in largest_docs]
```

We now come to the end of Section 1.1 – the Document Ranking step. We now move on to the next section – where we talk about the Sentence Extraction.

Sentence Extraction

This step consists of 5 subparts.

Splitting the documents to the sentence level

Since we are now going to work at the sentence level, we take all the sentences in the ranked documents and split them, sentence by sentence. In the previous corpus text file, we had each document as one line in the text file. But now in our new file, we have one sentence forming each line in our text file. Essentially, the documents in this step are in fact, the sentences. We will do this in the setup stage of the code and have one separate text file for each document in the corpus with the text file having all the sentences in the document, line by line. In the live stage, we just load back the relevant text file's sentences back into a data-structure that holds each sentence and also tagged with the document from which the sentence came from. This will be useful for the future steps. Also note that this section is streamed and the relevant text files are opened and sentences extracted during the live stage.

```
file1 = open("cmptext.txt","r")
stemmer = PorterStemmer()
for document in file1:
    spreprocessed = []
    doc_num += 1
    for line in document.split('. '):
        temp1 = []
        temp2 = []
        temp1 = gensim.utils.simple_preprocess(line, max_len = 20)
        for word in temp1:
            if word not in stop_words:
                temp2.append(word)
        spreprocessed.append(stemmer.stem_documents(temp2))
    with open("spreprocessed" + str(doc_num) + ".txt", "w+") as fp:
        pickle.dump(spreprocessed,fp)
    fp.close()
    del spreprocessed[:]
file1.close()
```

Constructing some familiar data-structures

We will now create some familiar data structures that we have already seen in Section 1.1. We will need a new vocabulary and this vocabulary will consist of only the words and phrases in our expanded query. So now we have a new vocabulary.

We need a new inverted index built on these vocabulary terms, and we build this in the same way we did before. This index is built on the sentences we extracted, considering each sentence as a separate document. Finally, we need a final set of relevant sentences to do the scoring on. We extract these in the very same way that we extracted the relevant documents. All these activities are conducted on the data structure we created in Section 1.2.1 with the sentences and the tagged document numbers.

Also note that this section is not streamed any further. We use the data structure created in Section 1.2.1. It is quite clear that most of these steps follow the steps in Section 1.1. For these reasons, we do not venture into an extremely detailed explanation about the steps. In fact, most of the functions from the Document Ranking step are reused here.

```
vocabulary2 = list(expanded_query)
inverted_index2= get_inverted_index_query_terms(preprocessed_tuple, vocabulary2)
relevant_sent = get_relevantdocs(expanded_query, inverted_index2)
```

The BM-25 Scoring System

It is now time to score the sentences we have extracted in the relevant sentences data structure that we created. We will not use the TF-IDF scoring system again. Rather, we move on to a more powerful scoring system that might be more appropriate for small “documents”, which in our case, is the sentences. This method could’ve been used for the Document Ranking step as well, but TF-IDF does the job well for documents and has been widely used for that purpose as well. We will now talk about the BM-25 scoring system briefly before we explain how we calculated the scores in our application.

In Information Retrieval, Okapi BM25 (BM stands for Best Matching) is a ranking function used by search engines to rank matching documents according to their relevance to a given search

query. It is based on the probabilistic retrieval framework developed in the 1970s and 1980s by Stephen E. Robertson, Karen Spärck Jones, and others. The name of the actual ranking function is BM25. To set the right context, however, it is usually referred to as "Okapi BM25", since the Okapi information retrieval system, implemented at London's City University in the 1980s and 1990s, was the first system to implement this function. This is a Bag of Words model as well. This model does not use the concept of vector spaces; it is just the direct implementation of a formula. There are many variations to the formula but the one shown below is the most widely used variation and we have followed that as well. For more details about this model, please visit their Wikipedia page.

Given a query Q, containing keywords $q_1, \ldots, q_n$, the BM25 score of a document D is:

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)},$$

$$\text{IDF}(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5},$$

where N is the total number of documents in the collection, and $n(q_i)$ is the number of documents contai

```
def get_bm25(relevant_sent, inverted_index, expanded_query, preprocessed_tuple):
    lengths = [len(sentence_tuple[1]) for sentence_tuple in preprocessed_tuple]
    bm25_scores = dict((el,0) for el in relevant_sent)
    avg_dl = sum(lengths)/len(lengths)
    number_of_sent = len(preprocessed_tuple)
    idf_words = {word:0 for word in expanded_query}
    i = 0
    for word in expanded_query:
        if word not in inverted_index.keys():
            idf_words[word] = log(float(number_of_sent + 0.5)/(0.5),10)
            continue
        idf_words[word] = log(float(number_of_sent - len(inverted_index[word])+ 0.5)/(len(inverted_index[word]) + 0.5),10)

    for sent_number in relevant_sent:
        for word in expanded_query:
            bm25_scores[sent_number] += float(idf_words[word]*preprocessed_tuple[sent_number-1][1].count(word)*2.5)/(preprocessed_tuple[sent_number-1][1].count(
    return bm25_scores
```

Sentence Ranking

This step is very similar to the final Document Ranking step. Now that we have all the relevant sentences and their corresponding scores, all we need to do is throw them into a heap and extract the 100 most relevant sentences. Again, we have chosen to use 100, but the number can be increased/ decreased based on the application. Now we have the 100 most relevant sentences from our huge corpus that we had initially, with respect to the query given.

```
heap_sentences = [(-value, key) for key,value in scores_bm25.items()]
largest_sentences = heapq.nsmallest(100, heap_sentences)
largest_sentences = [(key, -value) for value, key in largest_sentences]
```

Pretty Printing

The idea is to print the results neatly in such a way that the main answer i.e. the most relevant sentence comes right at the top and below that we have the next most relevant sentence and its corresponding document and so on. If a particular document has already been listed, we don't consider sentences from that particular document again. Therefore, we want to print

only the most relevant sentence in each document. If we find for example that all the 100 relevant sentences, say appear in just 3 documents, we rank and display the remaining relevant documents based on the Document Ranking order. This is just to say that the Sentence Extraction order takes precedence over the Document Ranking order.

This step does not involve any major algorithm of any sorts, so we do not go into any details here. At the end of this step, we have a "Google Search engine"-esque result structure with one relevant sentence under each document and one main answer right at the top. Note here that "the most relevant sentence" in a document is the sentence with the highest score in that document.

```
sentenced_docs = set()
for sentence in [x[0] for x in largest_sentences]:
    sentenced_docs.add(preprocessed_tuple[sentence-1][0])
sentenced_docs_copy = [x for x in sentenced_docs]
print '*************************************************************'
index_ans = get_index_ans(preprocessed_tuple, sentence_number = largest_sentences[0][0])
fp = open('cmptext.txt', 'rb')
doc = 1
for line in fp:
    if doc == preprocessed_tuple[largest_sentences[0][0]-1][0]:
        print 'MAIN ANSWER : ' + str(line.split('. ')[index_ans])
    doc += 1
fp.close()

print '*************************************************************'
for sentence in [x[0] for x in largest_sentences]:
    doc_number_of_sentence = preprocessed_tuple[sentence-1][0]
    if doc_number_of_sentence in sentenced_docs:
        print 'Document ' + str(doc_number_of_sentence)
        index_ans = get_index_ans(preprocessed_tuple, sentence_number = sentence)
        fp = open('cmptext.txt', 'rb')
        doc = 1
        for line in fp:
            if doc == doc_number_of_sentence:
                print 'SENTENCE : ' + str(line.split('. ')[index_ans])
            doc += 1
        fp.close()
        sentenced_docs.remove(doc_number_of_sentence)
    else:
        continue

for document in [x[0] for x in largest_docs]:
    if document not in sentenced_docs_copy:
        print 'Document ' + str(document)
```

With this, we come to the end of Section 1.2 and our discussion on the Simple Search engine model. In the next section, we discuss some shortcomings of this system and the need for novel techniques.

Some additional features

We decided to add some additional features to this search engine to make it more robust and more of a software than a simple programme. As of now, the system was running on a terminal and printing the results on a terminal. The idea was to move it to a web server, where we would host it and users could use it by connecting to the IP address. We used a JSON-RPC server for this.

It uses JSON, which is a lightweight data-interchange format. The idea was to create a client file, a simple html file with the requisite GUI and a server file which would access the python code. The data would be transferred between the server and the client in JSON format. The query would be transferred from the client side to the server side in JSON format and we would return the results in appropriate JSON format back to the client side. The GUI was quite simple and resembled any conventional search engine.

```
@dispatcher.add_method
def run_search_engine(input_query):
        return QAlive2.live(model, input_query, vocabulary, length_preprocessed, inverted_index, document_dictionary, norms, nod,
list_of_document_tfidf_dicts)

@dispatcher.add_method
def autocomplete(input_query):
        return search_bar.browser_main(input_query, my_trie, spell_vocabulary, spellchecker)


@Request.application
def application(request):
        response = JSONRPCResponseManager.handle(request.data, dispatcher)
        return Response(response.json, mimetype='application/json', headers={'Access-Control-Allow-Headers':['access-control-allow-origin',
'content-type'],'Access-Control-Allow-Origin':'*'})



if __name__ == '__main__':
        model = gensim.models.Word2Vec.load('vocab.txt')
        vocabulary = list()
        length_preprocessed = 0
        inverted_index = dict()
        vocabulary = model.wv.vocab.keys()
        fp = open("preprocessed.txt", "rb")
        length_preprocessed = pickle.load(fp)
        fp.close()
        fp = open("inverted-index.txt", "rb")
        inverted_index = pickle.load(fp)
        fp.close()
        fp = open('my-trie.txt', 'rb')
        my_trie = pickle.load(fp)
        fp.close()
```

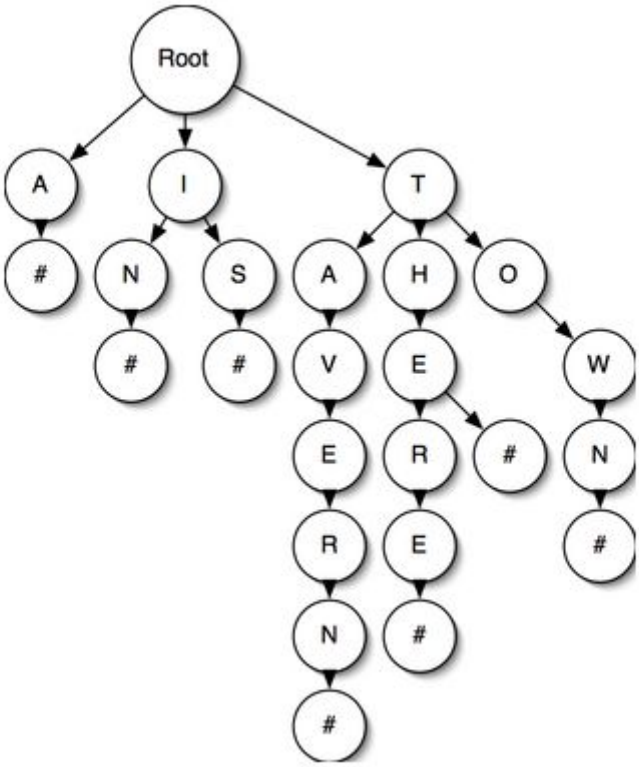
```
        fp = open('unstemmed-words.txt', 'rb')
        spell_vocabulary = defaultdict(int)
        for line in fp:
                spellcheck.train(spell_vocabulary, spellcheck.words(line))
        fp.close()
        spellchecker = spellcheck.SpellChecker(spell_vocabulary)
        with open("spelling-vocab.txt", "rb") as fp:
                my_words = pickle.load(fp)
        fp.close()
        with open("document-index.txt", "rb") as fp:
                document_dictionary = pickle.load(fp)
        with open('norms.txt', 'rb') as fp:
                norms = pickle.load(fp)
        with open('tfidf-scores.txt', 'rb') as fp:
                nod = pickle.load(fp)
                list_of_document_tfidf_dicts = pickle.load(fp)
        fp.close()
        run_simple('10.19.1.166', 4000, application)
```

Now that the application wasn't running on the terminal anymore, we could add some features that we couldn't before. The first feature we would add was a spell-checker. This was a feature that corrected incorrect spellings made by the user to correct ones. It was a very simple model. The programme simply corrected spellings of words by making a maximum of two edits (if it found that the spelling was initially incorrect). An edit was defined as either, adding a character, deleting a character or switching two adjacent characters. If there is more than one word that can be formed by this process, then it is corrected to the word that occurs more frequently. Hence it was a simple probability based approach that we were following here.

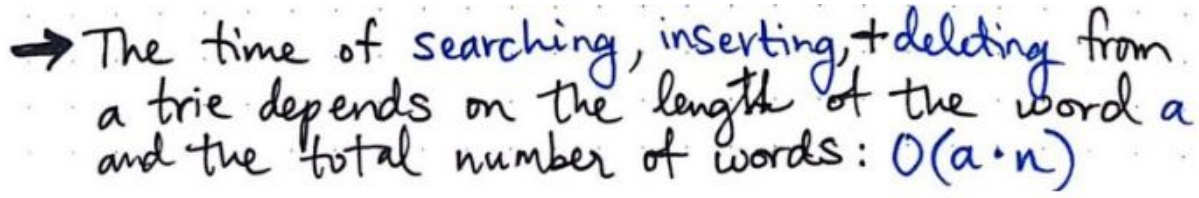
```
class SpellChecker(object):

    def __init__(self, vocabulary):
        self.vocabulary = vocabulary

    alphabet = 'abcdefghijklmnopqrstuvwxyz'

    def _edits1(self, word):
        splits = [(word[:i], word[i:]) for i in range(len(word) + 1)]
        deletes = [a + b[1:] for a, b in splits if b]
        transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b) > 1]
        replaces = [a + c + b[1:] for a, b in splits for c in self.alphabet if b]
        inserts = [a + c + b for a, b in splits for c in self.alphabet]
        return set(deletes + transposes + replaces + inserts)

    def _known_edits2(self, word):
        return set(e2 for e1 in self._edits1(word) for e2 in self._edits1(e1) if e2 in self.vocabulary)

    def _known(self, words):
        return set(w for w in words if w in self.vocabulary)

    def correct_token(self, token):
        candidates = self._known([token]) or self._known(self._edits1(token)) or self._known_edits2(token) or [token]
        return max(candidates, key = self.vocabulary.get)

    def correct_phrase(self, text):
        tokens = text.split()
        return [self.correct_token(token) for token in tokens]
```

```
def words(text):
        return re.findall('[a-z]+', text.lower())

def train(model, features):
        for f in features:
            model[f] += 1
```

The next feature we added was an autocomplete feature. This was essentially a feature in the search bar that completed user's queries for them. We use a data structure called trie for this. Tries are widely used for this exact process. They are essentially trees of the form given in the image below.



As you can see in the image above, words can be stored in the trie in a tree form and one can traverse through the trie to find words. We use this exact property of the trie to perform our autocompletes. As the user keeps typing letters of his/her query, we keep traversing down the trie to give the different possibilities of completion.



```
class TrieNode(object):
    """
    Our trie node implementation. Very basic. But does the job
    """

    def __init__(self, char):
        self.char = char
        self.children = []
        # Is it the last character of the word.
        self.word_finished = False
        # How many times this character appeared in the addition process
        self.counter = 1


    def add(self, word):
        """
        Adding a word in the trie structure
        """
        node = self
        for char in word:
                found_in_child = False
                # Search for the character in the children of the present `node`
                for child in node.children:
                        if child.char == char:
                                # We found it, increase the counter by 1 to keep track that another
                                # word has it as well
                                child.counter += 1
                                # And point the node to the child that contains this char
                                node = child
                                found_in_child = True
                                break
                # We did not find it so add a new child
                if not found_in_child:
                        new_node = TrieNode(char)
                        node.children.append(new_node)
                        # And then point node to the new child
                        node = new_node
        # Everything finished. Mark it as the end of a word.
        node.word_finished = True


    def find_prefix(self, prefix):
        """
        Check and return
        1. If the prefix exists in any of the words we added so far
        2. If yes then how may words actually have the prefix
        """
        node = self
        # If the root node has no children, then return False.
        # Because it means we are trying to search in an empty trie
        if not self.children:
                return False, 0
        for char in prefix:
                char_not_found = True
                # Search through all the children of the present `node`
                for child in node.children:
                        if child.char == char:
                                # We found the char existing in the child.
                                char_not_found = False
                                # Assign node as the child containing the char and break
                                node = child
                                break
                # Return False anyway when we did not find a char.
                if char_not_found:
                        return False, 0
```

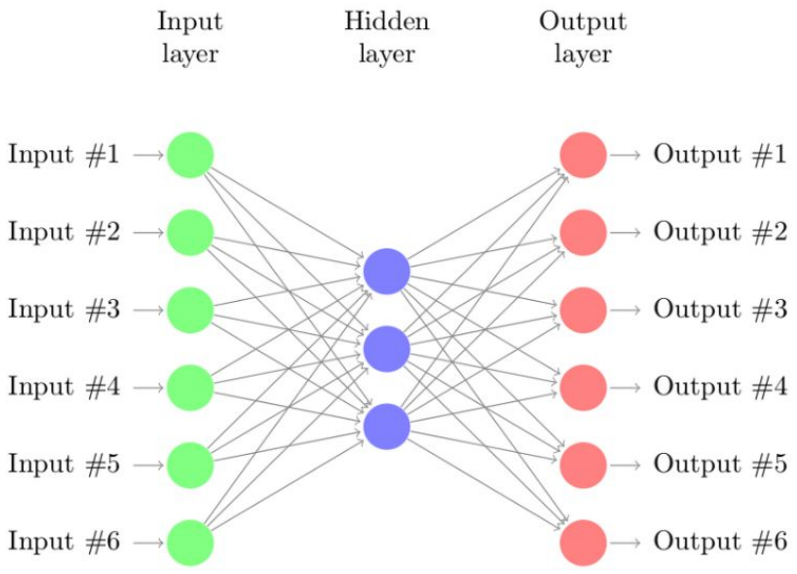
```
        # Well, we are here means we have found the prefix. Return true to indicate that
        # And also the counter of the last node. This indicates how many words have this
        # prefix
        return True, node.counter

    def all_suffixes(self, prefix):
        results = set()
        if self.word_finished:
            results.add(prefix)
        if not self.children:
            return results
        return reduce(lambda a, b: a | b,  [node.all_suffixes(prefix + node.char) for node in self.children])

    def autocomplete(self, prefix):
        node = self
        for char in prefix:
            if char not in [child.char for child in node.children]:
                return None
            for child in node.children:
                if child.char == char:
                    node = child
        return list(node.all_suffixes(prefix))
```

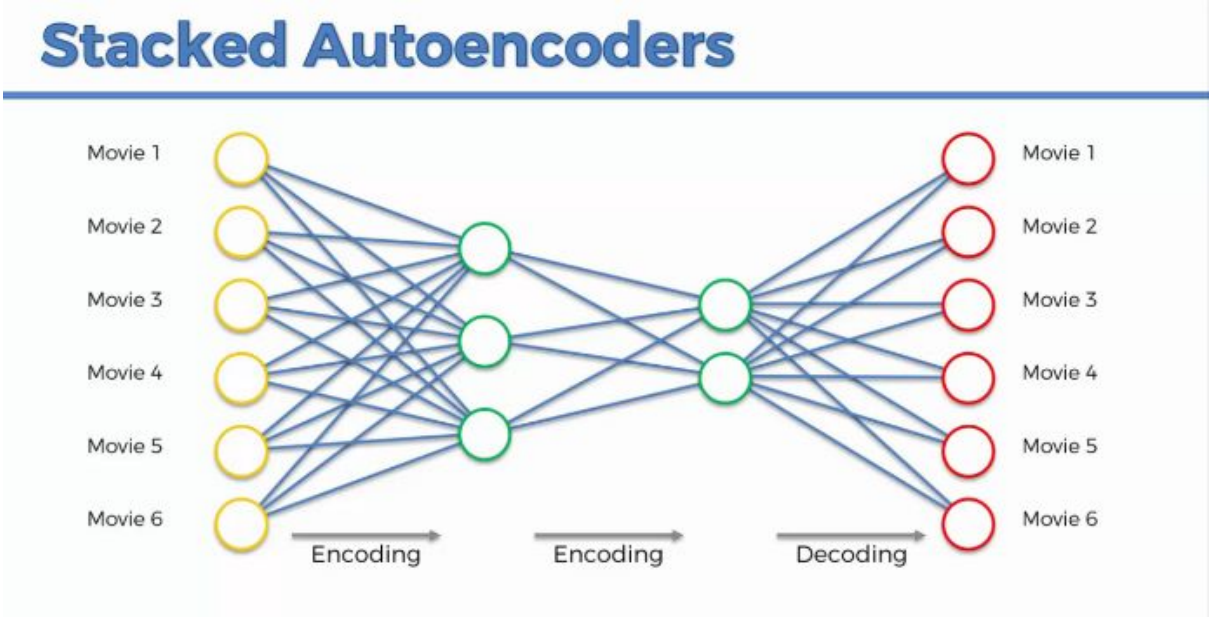
We then combined the spell-check feature and the auto-complete feature to build the final search bar. A little bit of basic HTML and CSS and the feature was ready to be deployed.

```
def check_spellings(spellchecker, input_query):
        return ' '.join(spellchecker.correct_phrase(input_query))


def automatic_complete(spellchecker, my_trie, input_query, my_words):
        if my_trie.autocomplete(input_query) is not None:
                if len(my_trie.autocomplete(input_query)) >=10:
                        return my_trie.autocomplete(input_query)[0:10]
                return my_trie.autocomplete(input_query)
        input_query = check_spellings(spellchecker, input_query)
        list_of_suggestions = set()
        if my_trie.autocomplete(input_query) is None:
                tokens = input_query.split(' ')
                last_three_words, last_two_words, last_word = [], [], []
                if len(tokens) >= 3:
                        last_three_words = tokens[len(tokens)-3: len(tokens)]
                if len(tokens) >= 2:
                        last_two_words = tokens[len(tokens)-2: len(tokens)]
                last_word = tokens[len(tokens)-1]
                if last_three_words:
                        if my_trie.autocomplete(' '.join(last_three_words)) is None:
                                for word in my_words:
                                        if last_three_words[0] in word and last_three_words[1] in word and last_three_words[2] in word:
                                                list_of_suggestions.add(' '.join(tokens[0: len(tokens)-3]) + ' ' + word.replace('_',' '))
                                                if len(list_of_suggestions) > 10:
                                                        break
                        else:
                                list_of_suggestions.update([' '.join(tokens[0: len(tokens)-3]) + ' ' + word for word in my_trie.autocomplete('
'.join(last_three_words))])
                if last_two_words and not list_of_suggestions:
                        if my_trie.autocomplete(' '.join(last_two_words)) is None:
                                for word in my_words:
                                        if last_two_words[0] in word and last_two_words[1] in word:
                                                list_of_suggestions.add(' '.join(tokens[0: len(tokens)-2]) + ' ' + word.replace('_',' '))
                                                if len(list_of_suggestions) > 10:
                                                        break
                        else:
                                list_of_suggestions.update([' '.join(tokens[0: len(tokens)-2]) + ' ' + word for word in my_trie.autocomplete('
'.join(last_two_words))])
                if last_word and not list_of_suggestions:
                        if my_trie.autocomplete(last_word) is None:
                                for word in my_words:
                                        if last_word in word:
                                                list_of_suggestions.add(' '.join(tokens[0: len(tokens)-1]) + ' ' + word.replace('_',' '))
                                                if len(list_of_suggestions) > 10:
                                                        break
                        else:
                                list_of_suggestions.update([' '.join(tokens[0: len(tokens)-1]) + ' ' + word for word in my_trie.autocomplete
(last_word)])
                else:
                        list_of_suggestions.update(my_trie.autocomplete(input_query))
                        #Try to complete with last few words here ?
                if len(list_of_suggestions) > 10:
                        return list(list_of_suggestions)[0:10]
                else:
                        return list(list_of_suggestions)
```

The next feature, which would be the last one, was quite an interesting one. It was based on an idea to bring documents that a user may like more to the top of the rankings list for a particular query. This would involve the original document ranking scores based on TF-IDF and also some scores based on how relevant a document would be to the particular user. We would also be suggesting queries to the user in a similar fashion. The autocomplete feature would suggest to the user, queries that they were more likely to prefer. We used a Recommender system for this purpose. I won't get too much into the details of a Recommender system here but it is

essentially a system that makes suggestions and recommendations to users based on what they have already seen/liked. It is usually used to suggest songs and movies to users but here we would be using it to suggest documents to users based on documents they had seen before. This system can be both memory-based and model-based. A memory-based model uses some similarity metrics to identify similarities between users and between the documents as well, very similar to the vector space model. A model-based approach only tries to identify some particular patterns in the data and then tries to suggest some documents to the user. It involves some Machine Learning to recognize the patterns in data. The memory-based approach is in general, said to be more accurate but slower. The model-based approach is faster but in general it is said to be less accurate. But there are some very good models that have achieved accuracies better than the best memory-based models. Hence, the words “in general” are quite important. Now coming to our particular system, we went for a model-based approach. There are several model-based approaches available but we went for a slightly novel and off-beat approach using Neural Networks. Neural networks have been rising in popularity over the last few years and they have been used for multiple different tasks, with spectacular success. They can be used for supervised and unsupervised learning and there are many different types of neural networks available for each task. For our particular requirement, we use a type of network called a Stacked Autoencoder. This actually falls under the category of self-supervised learning but we list it under unsupervised learning for convenience. The reason for listing it in this manner will be clear in the next few sentences. Generally, for supervised learning, there is labelled data, meaning each item in the dataset will be tagged with a label - and the type of label varies depending on whether it is a Regression setting or a Classification setting. Here, we have no labelled data and hence, automatically, we fall into the unsupervised learning category. The reason we call it self-supervised learning is because an autoencoder takes the input and tries to regenerate the input itself as the output. So in a way, the input itself is the label for the date. The system tries to regenerate the input itself. The way it does this is that, it encodes the input vector into a vector of smaller dimensions and then decodes this into the output vector, which is of same dimensions as the input vector. The diagrams below explain this quite well.





We do not intend to go into the details of the autoencoder too much here. It is quite complex and involves a lot of math. There are different types of Autoencoders, like the Stacked Autoencoder and the Deep Autoencoder. These often get confused with each other but they are not the same. A stacked autoencoder (SAE) is what we will use for our purpose and is essentially an autoencoder with multiple encoding and decoding layers. For example, our particular autoencoder uses two encoding and two decoding layers, to preserve symmetry. This is apart from the input and output layers. A deep autoencoder is completely different and is actually Deep Boltzmann machines stacked one on top of the other. We will not go into the details at this point. The main take-away is that autoencoders are neural networks that are capable of encoding the input data into a vector of smaller size and then decoding it back to a vector of the same size such that the input matches the output as much as possible. Now to explain things quite intuitively, each input corresponds to a user and each user has a vector of size equal to the number of documents in our corpus. The value of each element of this vector is the number of times the user has opened that particular document (including zeroes). The number of times the user has opened a document represents how much the user likes that document. The task of this recommender system is that after training, it should be able to predict how much the user would like the documents he/she hasn't seen before i.e. the 0 values. Using these, we recommend new documents to the user. The input size is therefore, the total number of documents and so is the output size. The hidden layers are in our control and as mentioned earlier, we have chosen a 20,10,10,20 setting after some parameter tuning. So we have an input layer, four hidden layers and an output layer in our neural network and this will encode the input vector and decode it and along the way, find some patterns in the data. There is some pre-processing to do before we start. We need to prepare the dataset. We create a function that updates a MySQL database and populates it with user and document data. We then transfer data from this database to training and test data to run the autoencoder. Essentially, from the client side, as the user clicks on a document, the data is sent back to the server side and updated in the database. At the end of the day, perhaps, the data is transferred to training and test dataset and this is used to train the neural network. Now the final task is to combine the tf-idf scores and the scores from the recommender system. After some parameter tuning and having made some empirical judgements, we take a 90% weightage for the tf-idf scores and a 10% weightage for the recommender system scores. The final score is taken into account to calculate the top 10 documents, as described in section 1.1.7

.

```
#Getting the data
training_set = pd.read_csv('q1.base', delimiter = '\t')
test_set = pd.read_csv('q1.test', delimiter = '\t')
training_set = np.array(training_set, dtype = 'int')
test_set = np.array(test_set, dtype = 'int')


#Getting total number of users and movies
nb_users = 100
fp = open('document-index.txt', 'rb')
doc_index = pickle.load(fp)
fp.close()
nb_documents = len(doc_index)
min_user_index = 1
max_user_index  = 100


#Creating User-Movie matrix
def convert(data):
        new_data = []
        global doc_index
        doc_ids = doc_index.values()
        global min_user_index
        global max_user_index
        for id_users in range(min_user_index, max_user_index + 1):
                id_movies = data[:,1][data[:,0] == id_users]
                id_ratings = data[:,2][data[:,0] == id_users]
                ratings = np.zeros(nb_documents)
                ratings[np.array([doc_ids.index(str(i)) for i in id_movies], dtype = 'int')] = id_ratings
                new_data.append(list(ratings))
        return new_data

training_set = convert(training_set)
test_set = convert(test_set)
```

```
#Converting data to Torch tensors
training_set = torch.FloatTensor(training_set)
test_set = torch.FloatTensor(test_set)

#Architecture of the Stacked AutoEncoder
class SAE(nn.Module):
        def __init__(self,):
                super(SAE, self).__init__()
                self.fc1 = nn.Linear(nb_documents, 20)
                self.fc2 = nn.Linear(20, 10)
                self.fc3 = nn.Linear(10, 20)
                self.fc4 = nn.Linear(20, nb_documents)
                self.activation = nn.Sigmoid()
        def forward(self, x):
                x = self.activation(self.fc1(x))
                x = self.activation(self.fc2(x))
                x = self.activation(self.fc3(x))
                x = self.fc4(x)
                return x

sae = SAE()
criterion = nn.MSELoss()
optimizer = optim.RMSprop(sae.parameters(), lr = 0.01, weight_decay = 0.5)

#Training the SAE
nb_epoch = 200
for epoch in range(1, nb_epoch + 1):
        train_loss = 0
        s = 0.
        for id_user in range(nb_users):
                input_data = Variable(training_set[id_user]).unsqueeze(0)
                target = input_data.clone()
                if torch.sum(target.data > 0) > 0:
                        output = sae.forward(input_data)
                        target.require_grad = False
                        output[target == 0] = 0
                        loss = criterion(output, target)
```

```
                        mean_corrector = nb_documents/ float(torch.sum(target.data > 0) + 1e-10)
                        loss.backward()
                        train_loss += np.sqrt(loss.data[0]*mean_corrector)
                        s += 1
                        optimizer.step()
        print 'epoch: ' + str(epoch) + ' loss: ' + str(train_loss/s)

#Testing the SAE
test_loss = 0
s = 0.
for id_user in range(nb_users):
        input_data = Variable(training_set[id_user]).unsqueeze(0)
        target = Variable(test_set[id_user]).unsqueeze(0)
        if torch.sum(target.data > 0) > 0:
                output = sae.forward(input_data)
                target.require_grad = False
                output[target == 0] = 0
                loss = criterion(output, target)
                mean_corrector = nb_documents/ float(torch.sum(target.data > 0) + 1e-10)
                test_loss += np.sqrt(loss.data[0]*mean_corrector)
                s += 1
print 'test loss: ' + str(test_loss/s)

torch.save(sae, 'my_sae.pt')
```

Drawbacks of the Simple Search Engine

The following are some of the drawbacks of the model described above

- The model can only narrow the search down to a single sentence, but cannot extract the exact answer to the query from the sentence or from the corpus. It is not exactly a Question-Answering model.
- It is a Bag of Words model which means all semantic structure is lost. It simply works on the basis of word frequencies. Such a model will not be able to understand, semantically, what the user is trying to ask in his/ her query
- The model can only reach certain levels of accuracy, although that accuracy will increase with the size of the corpus.
- The model can only reach certain levels of speed since it is, at the end of the day, a quite primitive and computationally intensive model. The speed is not accepted to increase or decrease in the live part of the code, but the time to process the setup code will increase with the size of the corpus. A better system might ensure that the query processing stage runs faster as well.

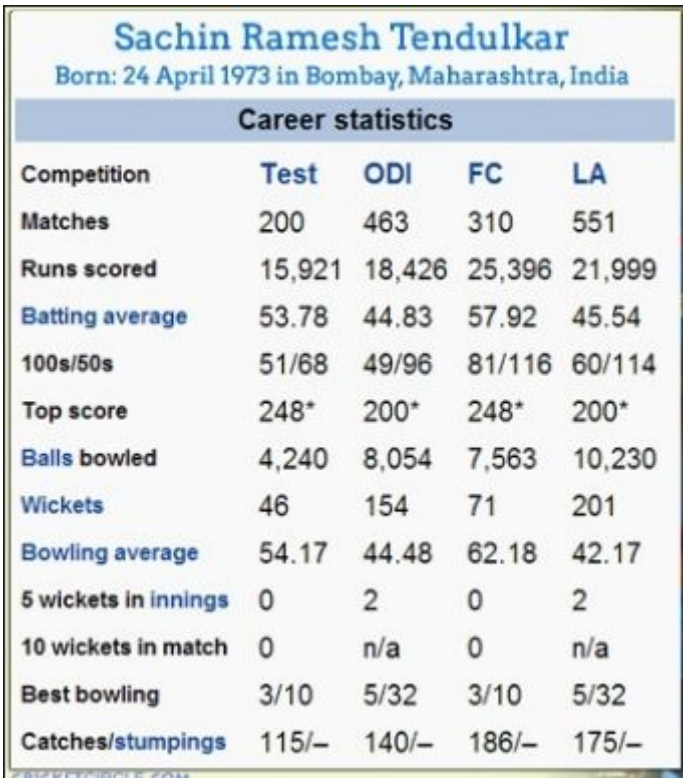
As we can see, this model definitely has its drawbacks and for these reasons, we need to move on to more novel techniques. It is extremely difficult to extract just the answer from a corpus/ sentence without using Natural Language Processing techniques and that is where we will go ahead with in this paper. There are other methods like Latent Dirichlet Allocation and Hierarchical Latent Dirichlet Allocation that are extremely powerful algorithms that may give better results in the document ranking and sentence extraction steps. But there is no replacement for some semantic analysis techniques to pick out the final answer from the extracted sentence or from the entire corpus. In the next section, we discuss one particular Natural Language Processing technique called Entity Profiling and how it could help in creating a database of entities and corresponding information at the corpus level. This database can then be used to directly pick out answers to factoid based queries. This is a novel technique that hasn't been done before and we attempt this with the hope that it will improve the Question-Answering capabilities of our system. Section 2 of this paper is all about the task of Entity Profiling and how it can be applied in this project.

3. AN APPROACH TO ENTITY PROFILING

The task at hand now, was to try and extract exact answers to given queries. We have seen how Google is able to answer questions like “What is the mileage of the new Honda Amaze?” or “What is the capital of Canada?” directly by giving us a one word answer right at the top of the document ranking list. This is not an easy task and it requires a lot of analysis. These kinds of tasks often fall under the domain of Question Answering (QA), a task that people have been battling with for a long time. They have tried different approaches to tackle this task and one particular place for them to showcase their new ideas is the TREC (Text REtrieval Conference) annual Question answering track. Scientists come up with new ideas for Question answering and put their systems to the test in this annual event. There are many different question types, like factoid-based questions (“What is the temperature of the sun?”), list based questions (“List the different organs of the body”) , definition based questions (“What is voltage?”), biographical questions (“Who is Narendra Modi?”), scenario based questions and even ‘why’ types of questions. Most QA systems focus on the factoid-based questions, which are the easiest of all to handle. In fact, in our particular use-case as well, it seems like most queries that we will handle will be factoid-based. So the task is to handle factoid based questions and return answers. There are multiple different ways to do this and the approach proposed in this report is quite different and quite difficult as well. QA is one domain of Natural Language Processing and we will try to cross roads with another domain involving Attribute Extraction (AE) and Entity Profiling (EP).

The outline of the approach

Entity Profiling is the task of identifying entities of interest in structured/ unstructured text and extracting information about them. It is often seen that in many applications, we would like to extract information about a particular entity from a huge amount of structured/unstructured text data. An entity could be anything to which a proper noun maybe associated – Person, Location, Organization, Product, etc. This information that we wish to extract could be in the form of attribute-value pairs. This means that the onus is on the developer to come up with appropriate attributes for each entity type. For example, for a Person entity, some appropriate attributes could be Full Name, Occupation, Place of Work / Organization of affiliation, Date of Birth, Place of Birth and so on. For Location, we could have perhaps, latitude, longitude, area, population and so on. As we can see, for each entity type, there are different attributes associated with them. In most cases, these entities and these attributes are very domain specific. For example, we may not see the entity ‘Animal’ in a Software company related corpus. Our particular area of interest is the nuclear domain and hence we will have a very specific list of entities and a very specific list of attributes for each entity that we wish to extract. This has to be specified in prior – we must know clearly what we are looking for in the data. This has to be done with the help of a domain expert, who knows the corpus very well, to identify what the entities of interest are and what attributes we need to extract. This process is called Attribute Extraction (AE). Since, this is an extremely complex task; we will narrow down our entity list only to a Person entity, but more on this a bit later. Now assuming that we know the entities of interest and the attributes of interest, we wish to create an entity profile for all the different entities. An entity profile is essentially an attribute-value pair matching for each entity of the given entity type. The image below is a simple example of an entity profile.



Sachin Ramesh Tendulkar

Born: 24 April 1973 in Bombay, Maharashtra, India

Career statistics

Competition	Test	ODI	FC	LA
Matches	200	463	310	551
Runs scored	15,921	18,426	25,396	21,999
Batting average	53.78	44.83	57.92	45.54
100s/50s	51/68	49/96	81/116	60/114
Top score	248*	200*	248*	200*
Balls bowled	4,240	8,054	7,563	10,230
Wickets	46	154	71	201
Bowling average	54.17	44.48	62.18	42.17
5 wickets in innings	0	2	0	2
10 wickets in match	0	n/a	0	n/a
Best bowling	3/10	5/32	3/10	5/32
Catches/stumpings	115/–	140/–	186/–	175/–

Now if we can store this information corresponding to each entity in a database, having extracted the appropriate values for each attribute from the data, we have a ready-made entity profile for each entity in our corpus and we can easily answer questions based on these entities, if the questions are related to the attributes we have in our database. We can simply pick out the answer from the database and give it as a simple one word answer, much like how Google does. See the image above again and see how easy it would be to read off the database and answer a question like “What is Tendulkar’s Test batting average?”. So, we definitely understand the motivation to create such a database, since it makes the task of question answering so easy. Here is a rough outline of our task at hand.

1. Identify the entities (entity types) of interest (Person, Place, Organization, etc.)
2. Identify the attributes of interest for each entity (Full name, workplace, age, etc.)
3. Extract the attribute-value pairs from the text
4. Create a database for each entity type, each row will correspond to a separate entity. This structure however could be modified. For example, your entity type could be Location and your entities could be India, Pakistan and so on.
5. Given a query, identify the key terms and match them to corresponding rows and columns in our database, having identified which entity the query is about and what attribute-value pair the query is seeking. This can be done by direct-matching, hand-written rules or even supervised-learning.
6. Get the appropriate result from the database and return it to the user.

As we can see, most of the steps are quite clear. The biggest question mark is at step 3. How does one create such a database from raw text? This is the task we will try to attempt in the next subsection. We have not gone into the details too much here, because of the complexity of the task but we will try to give a brief account of the processes involved.

A semantic-based approach to entity profiling

There are several approaches to solve the problem of Entity Profiling, many of which are quite recent developments. These techniques often use Supervised and Semi-supervised learning techniques. There has also been a lot of application of neural networks in this area. But our approach is slightly different. It is a more semantic based approach that offers some significant benefits over the other approaches. It is based on a number of concepts in Natural Language processing (NLP) which allows the method to exploit the semantic information in the text. It allows for syntactic and structural variation and ensures extraction of attribute values at the cross-document and cross-sentence level. NLP has gained popularity again over the past few years and is being used widely for various applications. Hence, it is no surprise that it is being used for the task of Question Answering as well. We have used multiple research papers as references for writing the rest of this section but most of the ideas that will be presented here are from the paper : 'A semantic approach to cross-document person profiling in Web' written by Hojjat Emami, Hossein Shirazi and Ahmad Abdollahzadeh Barforoush. It is a rather long and complex paper that explains in detail, how one should look to perform the EP task for person entities in the web and tackles some very specific issues that other papers don't. We will borrow some simple ideas from the paper and present them in the rest of this section. Before we move on, we give a brief introduction to structured and unstructured text. Crudely, structured data is one that is in the form of short phrases that probably already directly contains the information we need. It does not follow standard writing norms. Unstructured data on the other hand, is how general text is written. It follows standard writing norms and we need specialized Information Extraction techniques to extract the attribute values from this data. For our particular purpose, we do not need to get into the web aspects of this paper and we also do not bother going into the structured data extraction aspect of this paper as we are fairly convinced that most of the data is going to be in the form of unstructured text.

Preprocessing

This is a rather long and important step. This is the condensed version of what we need for our particular application.

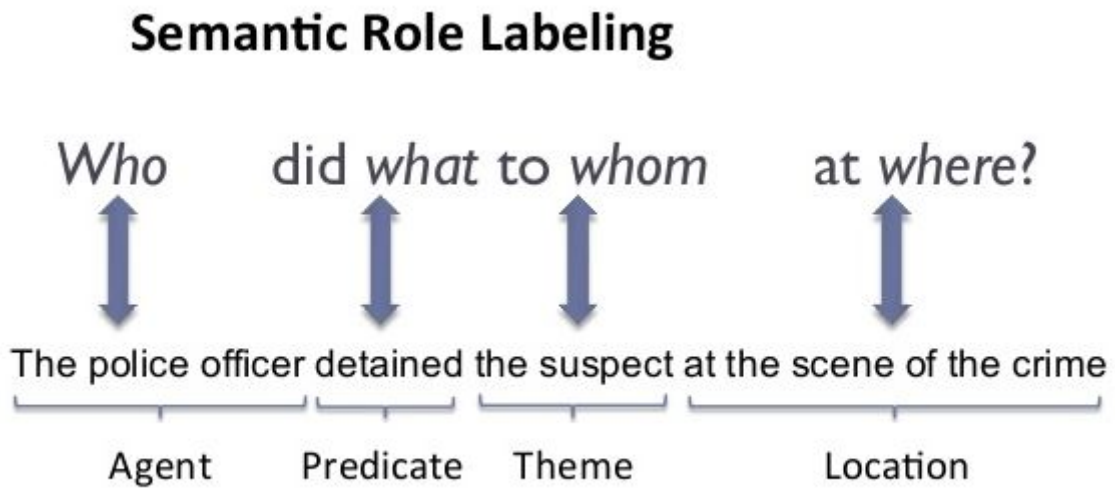
1) Named Entity Recognition: We use the Stanford Named Entity tagger to tag the entity types in sentences. We keep track not only of the entity types, but also maintain a unique index to distinguish the identity of the entity.
2) Coreference Resolution: This is an extremely difficult task that people are still working on. As of now, Stanford has a good coreference resolution system that we use to identify coreference resolution chains in the documents. A quick word on what coreference resolution is. It is essentially matching different references of the same entity to that particular entity. For example, in a piece of text, Barack Obama could be referred to as "President", "He" and so on. We need to identify that all of them match to the same person.

Semantic Analysis

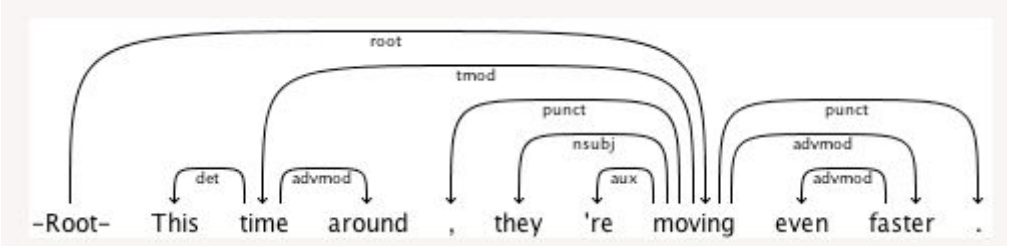
This is a rather long and complicated step as well. Once again, it is condensed to include only what we need for our particular application.

1) Word Sense Disambiguation: The task is to disambiguate each surface text word and entity mention to identify which of its senses is given in the text. For example, does the word "bank" refer to the river bank or the financial institution? This section solves this particular problem. In this way, the problems of polysemy and synonymy are solved as well. Without going into too many details, Babelfy, WordNet and DBPedia is used for this purpose.

2) Semantic Role Labelling: This is a crucial step and is extremely important in understanding the semantics in each sentence. In this step, we assign a "who did what to whom, when, why and how" structure. We use SENNA for this task and convert the resulting PropBank style roles to VerbNet style roles.



3) Dependency parsing: We use the Stanford Dependency parser for this purpose. This creates a typed dependency graph from the sentence. The words are the nodes and the edges represent word-word dependencies like "adjective" or "preposition" and so on.



4) Semantic Enrichment: This stage augments each dependency graph and semantic role frame with semantic information provided by the Word Sense Disambiguation step. Each disambiguated sense may cover a single word expression or multiple word expression in the dependency graph. To enrich semantic role frames with word senses, we simply replace each semantic role element with its corresponding disambiguated sense. The remainder elements that have no correspondence in disambiguated senses are left without mapping.

Attribute Extraction

This is the most important part of this section because this is where the main attribute-value pair extraction takes place. We will proceed with this section bearing in mind that we will only consider Person entities and we will use a set of 16 attributes, some of which are : Affiliation, Degree, Mentor, Phone, Award, Fax, Major, School, etc. This choice has nothing to do with our particular application and is only an example. There are different methods for attribute-value pair extraction for structured and unstructured text data. Since we have concluded that we will not encounter any structured data in our corpus, we will talk only about the unstructured text data extraction.

We identify that when it comes to unstructured text, most of the attribute values we are looking for appears in (i) verb constructions, expressed by either a main verb, which serves as an indicator to the attribute class (for example, the verb "born" indicates the attribute class Date of birth) or a light verb (like "is" or "was") (ii) or in noun based constructions that either appositives (for example, "the

Stanford professor, Jurafsky") or noun modifiers (for example, "American professor") to the given entity. Therefore, we need to use two different techniques, verb based AE and noun based AE.

1) Verb based AE: This step takes the semantic enriched role frames to extract discourse values for each attribute. Without going into too many details, we'd also like to add that Markov Logic Networks are used to extract these discourse values. These are joint models, which combine and make full use of probability and first order logic. A detailed discussion on these models are beyond the scope of this report. Now we have identified that some verbs ("born" for birth place) and special keywords ("professor" for Occupation) provide crucial clues to identify certain domain specific attributes of a person. How do we identify these seed trigger verbs and keywords for each attribute class ? We use some Supervised learning algorithms for this purpose. These algorithms can extract the seed trigger verbs and keywords corresponding to each attribute class. We present these four algorithms from the paper by H.Emami et al. Here for your reference. Note that in the Seed-Pruning step, the Pointwise Mutual Information (PMI) weighting schema is used to narrow down the seed list according to the top scoring terms. This PMI schema ensures that it gives importance to words that occur together but not occur together everywhere.

Algorithm 1: SEED EXTRACTION

procedure SEED-EXTRACTION(e, a, D)
Input:
e ▷ target person name
a ▷ target attribute
$D = \{d_1, \ldots, d_N\}$ ▷ training documents
Output:
Cv ▷ seed trigger verbs
Ck ▷ seed trigger keywords

foreach $d_i \in D$ **do**
  $M(e) \leftarrow$Co-reference(e, d_i)
  $d \leftarrow \{\bigcup_i s(e_i) \in d_i | \forall e_i \in M(e)\}$
  $\{V, W\} \leftarrow$SEED-VERB-EXTRACTION(e, a, d, D)
  $Cv \leftarrow Cv \cup \{V, W\}$
  $\{T, W\} \leftarrow$SEED-KEYWORD-EXTRACTION(e, a, d, D)
  $Ck \leftarrow Ck \cup \{T, W\}$
$Cv \leftarrow$ SEED-PRUNING(Cv)
$Ck \leftarrow$ SEED-PRUNING(Ck)
return Cv, Ck

Algorithm 2: CANDIDATE SEED VERB EXTRACTION

procedure SEED-VERB-EXTRACTION(e, a, d, D)
Input:
e ▷ target person name
a ▷ target attribute
d ▷ summarized document
$D = \{d_1, \ldots, d_N\}$ ▷ training documents
Output:
V ▷ candidate seed verbs
W ▷ candidate verbs' weights

```
1  V ← ∅
2  W ← ∅
3  foreach v ∈ V(a) do
4      foreach s in d do
5          if (e and v co-occur in s) then
6              G_d ← dependencyParse(s)
7              foreach verb in s do
8                  tag(e) ← dependencyTag(e, verb, G_d)
                     ▷ dependency tag of e with respect to
                     v in graph G_d
9                  if tag(e)=('sbj' or 'obj') then
10                     V ← V ∪ {verb}
11                     w ← Compute − Weight(e, verb) ▷
                         compute weight of verb by Eq.(4)
12                     W ← W ∪ {w}
13 return V, W
```

Algorithm 3: CANDIDATE SEED KEYWORD EXTRACTION

procedure SEED-KEYWORD-EXTRACTION(e, a, d, D)
Input:
e ▷ target person name
a ▷ target attribute
d ▷ summarized document
$D = \{d_1, \ldots, d_N\}$ ▷ training documents
Output:
T ▷ candidate seed keywords
W ▷ keywords' weights

```
1  T ← ∅
2  W ← ∅
3  foreach v ∈ V(a) do
4      foreach s in d do
5          if (e and v co-occur in s) then
6              foreach mention m in s do
7                  if (e and m co-refer in s) then
8                      T ← T ∪ {m}
9                      w ← Compute − Weight(e, m)   ▷
                         compute weight of m by Eq.(4)
10                     W ← W ∪ {w}
11 return T, W
```

Algorithm 4: SEED PRUNING

procedure SEED-PRUNING(Cs)
Input:
$Cs : \{T, W\}$ ▷ candidate seeds
▷ $T : \{t_1, t_2, ..., t_k\}$, candidate terms
▷ $W : \{w_1, w_2, ..., w_k\}$, terms' weights
Output:
S ▷ output seeds

```
W ← NORMALIZE-WEIGHT(W)
W ← SORT(W)
μ ← Σ_i w_i/|T|
T ← SELECT-TOPSCORE-TERMS(T, n) ▷ select n=30% of top score terms
foreach t_i ∈ T do
    if (w_i ≥ μ) then
        S ← S ∪ {t_i}
return S
```

Now, given that for each attribute class, we have our list of seeds associated with them, how do we find out which semantic role frames match with that particular class and entity ? We check for one of the following two conditions - (i) the verb predicate in the frame matches up with one of the seed verbs specific to the attribute, (ii) or one of the keywords or named-entities regarding the attribute appears in the arguments of the frame. After this, all we have to do is use some Markov Logic Network formulae to extract the actual values to the attributes we need, as we mentioned above. Again, we won't get into the details but some examples are given below.

w: Verb(v) ∧ Synonym(v, "born") ∧ SFrm(dID, v, "agent", e) ∧ SFrm(dID, v, "AM-Loc", g) ⇒ Birthplace(dID, e, g)

w: Verb(v) ∧ Synonym(v, "born") ∧ SFrm(dID, v, "agent", e) ∧ SFrm(dID, v, "AM-Loc", g) ∧ Location(dID, g) Person(dID, e) ⇒ Birthplace(dID, e, g)

w: Verb(v) ∧ Synonym(v, "win") ∧ SFrm(dID, v, "agent", e) ∧ SFrm(dID, v, "theme", y) ∧ Person(dID, e) ∧ AwardIndicator(dID, y) ∧ IsNounPhrase(dID, y) ⇒ Award(dID, e, y)

2) Noun based AE: This step is to extract attribute values which are found in noun constructions. For example, how do we find the occupation of a person from the sentence "The New York Mayor, Bloomberg signed the contract" ? We use the semantic boosted dependency graphs for this purpose and formulate the Markov Logic Networks AE rules. We already have our list of keywords for each attribute that we got from the Verb based AE step. An entity e and a keyword k in the graph is said to be related if (i) there is a dependency path of maximum length 3 between entity e and keyword k in which every dependency edge on the path is tagged with one of the following labels : *nsubj, nsubj-pass, appos, amod, nn, poss, prep_of, and rcmod* and if (ii) the keyword k and entity e co-refer. We formulate these conditions using Markov Logic Networks formulae. Some examples are given below. Note that we use an undirected version of the dependency graph as there is no guarantee that we find a path as described above in the directed graph.

(i) w: $DepRel(dID, \text{“amod”}, e, k) \wedge Co\text{-}refer(dID, e, k) \wedge Person(dID, e) \wedge NationalityIndicator(k) \Rightarrow Nationality(dID, e, k)$

(ii) w: $DepRel(dID, \text{“appos”}, e, m) \wedge DepRel(dID, \text{“amod”}, m, k) \wedge Co\text{-}refer(dID, e, k) \wedge Person(dID, e) \wedge NationalityIndicator(k) \wedge IsNounPhrase(m) \Rightarrow Nationality(dID, e, k)$

(iii) w: $DepRel(dID, \text{“nsubjpass”}, e, m) \wedge DepRel(dID, \text{“amod”}, m, k) \wedge Co\text{-}refer(dID, e, k) \wedge Person(dID, e) \wedge NationalityIndicator(k) \wedge IsNounPhrase(m) \Rightarrow Nationality(dID, e, k)$

It is to be noted that Tuffy is used to implement all the Markov Logic Network formulae. And with that, we come to the end of this particular section. As mentioned earlier, this is only an outline of the entire process as described in the paper by H.Emami et. al. We have not proposed an implementation for this process due to the inherent complexity. However having said that, hopefully this gives a good insight into the process of performing entity profiling from raw text for different entities. In the next and final section, we will conclude our work and also talk about some future work.

4. CONCLUSION AND FUTURE WORK

We have discussed a great many things in this report thus far and we will use this final section to summarise, conclude and talk about some future work and improvements. In the first section, we spoke about a simple search engine we made. We made use of some traditional Information retrieval techniques like the idea of the Vector space model. We used the TF-IDF scoring system to rank our documents and the BM-25 scoring system to extract our best sentences. We also used a recommender system using autoencoders to give documents that are more likely to be preferred by the user, a better score and hence a better ranking. We added some features in terms of a spell-checker and an autocomplete search bar as well. we wrapped the entire system with a JSON-RPC server and a neat HTML client with a GUI as well. We discussed some shortcomings of this system as well. In the second section, we spoke about entity profiling and described a method to perform EP on raw text. We used a semantic based approach for this purpose that involved many NLP techniques. We covered the different preprocessing steps, the semantic enrichment steps and the final attribute extraction step that covered verb based and noun based extraction techniques. With that we are done with the summary of this particular report.

This paper leaves many avenues un-explored and there is great scope for future work. Since we only implemented the first section of this paper and did not implement the second, our future work section will be based on the first half of this paper only. To begin with, we have used very primitive techniques in our search engine. There are better IR models and better scoring systems than TF-IDF and BM-25. Interested readers can try different models and scoring systems to achieve better results. We have used Word2Vec for our query expansion but there are many other techniques for this as well. Apart from improvements in accuracy, improvements in speed are always possible. One can try some parameter tuning with TF-IDF and BM-25 formulae or with the proportion of weightage given to the TF-IDF scores and the recommender system scores. Parameter tuning in the autoencoder section is possible as well. One can try to give appropriate recommendations for the autocomplete feature as well. Google does a great many things with their autocomplete feature and one can try to take inspiration and include any of their features into the system as well.

Thus, we have summarised and concluded our report and sincerely hope that you found that it was both an informative as well as an enjoyable read.

5. REFERENCES AND BIBLIOGRAPHY

[1] A semantic approach to cross-document person profiling in web by H.Emami et al.

[2] Text Mining for Product Attribute extraction by R.Ghani et al.

[3] Entity attribute extraction from Unstructured text with Deep belief Networks by Zhong et al.

[4] Extracting attributes of Named Entity from Unstructured text with Deep Belief Networks by Zhong et al.

[5] Markov Logic Networks for Natural Language Question Answering by T.Khot et al.

[6] A factoid question answering system using Answer Pattern matching by Nagehan et al.

[7] A survey of Answer Extraction Techniques in Factoid Question Answering by Mengqiu Wang

[8] An approach for extracting exact answers to Question Answering systems for english sentences by R.Barskar et al.

[9] Entity profiling in context with ontology by P. Moore

[10] Entity profiling by using Random Forest and trustworthiness of a source by S.Varma

[11] A new approach for Named Entity Recognition by B.Ertopcu et al.

[12] Implicit user profiling in News Recommender systems by Gulla et al.

[13] New classification framework to evaluate entity profiling on the web: past, present, future by Barforoush

[14] nlp.stanford.edu

[15] analyticsvidhya.com

[16] towardsdatascience.com

[17] kavita-ganesan.com (various blogs)

[18] radimrehurek.com/gensim

[19] github.com (various resources)

[20] Overview of TREC Question Answering tracks