

DOCUMENT SIMILARITY WITH SEMANTIC ANALYSIS

A Major Project Report

*submitted in partial fulfilment of the
requirements for the award of the degree
of*

Bachelor of Technology

in

COMPUTER ENGINEERING

Submitted

BY

YOGESH SAINI (1130402)

DIVYA BARDIYA(1130410)

PANKAJ KUMAR(1130512)



**DEPARTMENT OF COMPUTER ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY
KURUKSHETRA-136119, HARYANA (INDIA)**

April, 2017

TABLE OF FIGURES

1. Figure.1 :Flow Diagram.....	3
2. Figure.2:Start Page.....	5
3. Figure.3:Running Page.....	6
4. Figure.4:Output Page.....	6
5. Figure.5:Flow Diagram.....	16
6. Figure.6:Flow Diagram.....	30
7. Figure.7:Recall.....	32
8. Figure.8:F1Score.....	32
9. Figure.9:Precision.....	33
10. Figure.10:Input Story	33
11. Figure.11:Generated Titles.....	34
12. Figure. 12:Similarity Evaluation.....	34

TABLE OF CONTENTS

ABSTRACT

1	INTRODUCTION	1
1.1	Motivation	1
2	LITERATURE SURVEY	3
2.1	Summarization Based Approach	4
2.2	Statistical Approaches	5
2.3	Naive Based Approach	6
2.4	Rule Based Approach	7
3	DOCUMENT CLASSIFICATION	8
4	TITLE GENERATION	12
4.1	Extracting Keywords	12
4.1.1	Tagging	12
4.1.2	Normalization	13
4.1.3	Segmentaion	13
4.1.4	Stemming	13
4.1.5	Stopper Filtering	13
4.1.6	Finding Keywords	14
4.2	Maximum Entropy FrameWork	14
4.2.1	Feature Functions	14
4.3	Title Generation Model	15
4.3.1	Content Selection Model	16
4.3.2	Feature Set for Content Selection	17
4.3.3	Word/Part-of-speech feature	18
4.3.4	Positional Features	19
4.4	Word Translational Model	21
4.5	Title Synthesis Model	22
4.6	Decoding Algorithm	25
4.7	BLEU	27
4.8	Results	29

5	SOURCE CODE.....	32
6	CONCLUSION	75
	REFERENCES	76



CERTIFICATE

We hereby certify that the work which is being presented in the B.Tech. Major Project report entitled “**Document Similarity With Semantic Analysis**”, in partial fulfilment of the requirements for the award of the **Bachelor of Technology in Computer Engineering** is an authentic record of our own work carried out during a period from December 2016 to April 2017 under the supervision of Mr. **Mohit Dua, Assistant Professor**, Computer Engineering Department.

The matter presented in this project report has not been submitted for the award of any other degree elsewhere.

Signature of Candidate

Yogesh Saini (Roll.No. 1130402)

Divya Bardiya (Roll.No.1130410)

Pankaj Kumar (Roll.No. 1130512)

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Signature of Supervisor

Date:

Mohit Dua, Assistant Professor

Abstract

Classification is an automatic learning technique aimed at grouping a set of objects into subsets or clusters. The goal is to create clusters that are coherent internally, but substantially different from each other. In plain words, objects in the same cluster should be as similar as possible, whereas objects in one cluster should be as dissimilar as possible from objects in the other clusters. Automatic document classification has played an important role in many fields like information retrieval, data mining, etc. The approach is a completely different approach in which the words are clustered first and then the word cluster is used to cluster the documents. This reduces the noise in data and thus improves the quality of the clusters. In both these approaches there are parameters which can be changed according to the dataset in order to improve the quality and efficiency. This report explores the use of Support Vector Machines (SVMs) for learning text classifiers from examples. It analyzes the particular properties of learning with text data and identifies why SVMs are appropriate for this task. Empirical results support the theoretical endings. SVMs achieve substantial improvements over the currently best performing methods and behave robustly over a variety of different learning tasks. Title or short summary generation is an important problem in Text Summ - arization and has several practical applications. We present a discriminative learning fram - ework and a rich feature set for the title generation task. Secondly, we present a novel Bleu measure based scheme for evaluation of title generation models , which does not require human produced references . We achieve this by building a test corpus using the BBC news service. We propose two stacked log-linear models for both title word selection (Content Selection) and for ordering words into a grammatical and coherent title (Title Synthesis). For decoding a beam search algorithm is used that combines the two log-linear models to produce a list of k -best human readable titles from a news story. Systematic training and experimental results on the BBC-news test dataset demonstrate the success and effectiveness of our approach.

1. INTRODUCTION

Machine learning is a type of artificial intelligence (AI) that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can change when exposed to new data.

Learning from text and natural language is one of the great challenges of Artificial Intelligence and Machine Learning. One of the fundamental problems is to learn the meaning and usage of words in a data-driven fashion, that is from some given text corpus, possibly without further linguistic prior knowledge.

Classification is a division of data into groups of similar objects. Each group, called cluster, consists of objects that are similar between themselves and dissimilar to objects of other groups. In other words, the goal of a good document classification scheme is to minimize intra-cluster distances between documents, while maximizing inter-cluster distances (using an appropriate distance measure between documents). A distance measure (or, dually, similarity measure) thus lies at the heart of document classification. In classification, the classifier learns the association between objects and classes from a so-called training set.

Modelling the semantic similarity between text documents is an interesting problem for cognitive science, for both theoretical and practical reasons. Theoretically, it involves the study of a basic cognitive process with richly structured natural stimuli. Practically, search engines, text corpus visualizations, and a variety of other applications for filtering, sorting, retrieving, and generally handling text rely fundamentally on similarity measures. A special application is generating short summaries or titles from a given document. A title of a text, specially an article, is a succinct representation of relevant points of the input text. It differs from the task of producing abstracts, in the size of the generated text and focuses on the compressing the output. Titles are terse while abstracts are expressed using relatively more words. Automatic title generation tries to automate the process of providing more relevant or reflective insight into the input text rather than producing "catchy lines".

1.1 MOTIVATION:

Google claims to index over 2 billion web documents, and, over 1.5 million web documents are added to the World Wide Web every day. Human back-classification of at least 2 billion web documents would be virtually impossible, and even attempting to categorize all new web documents would require unacceptable human effort. One possible solution is to force web document authors to categorize each newly created page. This has three problems:

1. Web authors cannot be relied upon to categorize their pages correctly,
2. Authors are often prone to misclassify their documents in order to increase potential web traffic
3. It does not address the problem of the (at least) 2 billion web pages that have already been created.

The title of a text, especially a news article is a compact, grammatical and coherent representation of important pieces of information in the news article. Titles help readers to quickly identify information that is of interest to them. Although newspaper articles are usually accompanied by titles, there are numerous other types of news text sources, such as transcripts of radio and television broadcasts and machine translated texts where such summary information is missing.

2. LITERATURE SURVEY

There are many other potential applications and benefits that will accrue from being able to reliably and automatically cluster and categorize corpora of documents. Much of this document clustering work is based on using either supervised or unsupervised learning techniques in order to label particular web documents as belonging to a specific category, or grouping together similar documents into clusters. This research area closely overlaps with (and in recent times indistinguishable from) research efforts known as text classification and text categorization

Various techniques have been proposed that aim to develop accurate methods for autonomous categorization. However, the research literature in this area soon reveals that almost every single such proposed technique has been tested for its categorization accuracy using different datasets; any objective, scientifically sound comparison between two categorization techniques is therefore very difficult.

One of the technique is K-means which is described in [1]. The objective function of K-means is to minimize the average squared distance of objects from their cluster centers, where a cluster center is defined as the mean or centroid μ of the objects in a cluster C . As K-means has been implemented several times so we chose another technique i.e. probabilistic latent semantic described in [2]. Latent semantic analysis (LSA) is well-known technique which partially addresses these questions. The key idea is to map high-dimensional count vectors, such as the ones arising in vector space representations of text documents, to a lower dimensional representation in a so called latent semantic space. As the name suggests, the goal of LSA is to find a data mapping which provides information well beyond the lexical level and reveals semantical relations between the entities of interest. In contrast to standard LSA, its probabilistic variant has a sound statistical foundation and defines a proper generative model of the data.

Types of titles can be categorized into *indicative* (titles that identify the broader topic of the story) and *informative* (titles that identify the main event or purpose behind the story). Different methods for title generation typically handle generation of one or the other type of title. Most previous work on Title generation can be broadly categorized under Statistical, Rule-based and Summarization based (extractive) approaches. Below we discuss each of these approaches along with the pros and cons of each category. We also present a complete example of each category.

2.1 Summarization-Based Approaches

One way to connect summarization approaches with titles is to treat titles as summaries with a very short length. Given this, we can apply the methods of automatic text summarization to the task of automated title generation. In general, extractive approaches towards automatic text summarization can be categorized into three groups: surface-level approaches, entity-level approaches and the combinations of the two. The surface-level approaches find salient sentences for a summary using surface-level features including term frequency, the location in text, Cue phrases (i.e., phrases indicating the beginning of summary sentences such as “In conclusion”, “At the end”, etc.) and the number of key words or title words in a sentence. Several machine learning algorithms have been proposed for combining these surface level features. Naive Bayes, decision trees and semi supervised learning algorithms have been examined for combining features. There has been a consensus that, the location of the sentences and presence of cue phrases are more informative than other features. For example researchers have found that simply selecting the lead sentence of the news article as the title sentence, can be an effective strategy. The entity-level approaches include syntactic analysis, discourse analysis and semantic analysis[3]. These approaches rely heavily on the linguistic analysis of the source text to obtain linguistic structures such as discourse structure, syntactic structure and rhetorical structure to create a summary. There have also been efforts in combining surface level approaches and entity-level approaches together to produce better summaries. The advantage of Summarization approaches is that it alleviates the need to treat title generation as a special problem and one can simply take an existing text summarization system and request it to generate highly compressed summaries as titles. But the problem with resorting to summarization approaches for title generation is that, for summarization systems when the compression rate falls below 10%, the quality of generated summaries is poor. Since titles are typically no more than 10-15 words, the compression ratio is in fact far less than 10% for many news articles. This would mean that text summarization methods will create poor titles. Another problem with summarization approaches is that most of the techniques we discussed above are extractive in nature which constrains their use in title generation in other ways. For example: approaches that treat a full sentence as the minimum unit for a summary may result in longer than required titles. Another problem is that extractive techniques would pick only the phrases and words present in the article for inclusion in the title. But often we see that titles do not borrow the exact same words as present in the news article. Finally, a scenario where

extractive summarization approaches are not suitable is cross-lingual title generation in which news articles are present in one language and titles need to be generated in a different language. But statistical or corpus based techniques can be used without any specific changes for cross-lingual title generation just as they are used in the routine scenario. Cross-lingual title generation can indeed be very useful in cases where say a native language A (English) speaker is looking for language B (French) news articles on a specific topic or event. In such scenarios, the language A (English) speaker can identify the appropriate language B titles (French) if corresponding titles were available in English.

2.2 Statistical Approaches

Statistical or learning approaches assume the availability of a large training corpus (title news article pairs) and work in a supervised learning setting. The system or model is trained to learn the correlation between news articles and corresponding titles and then the learnt model is applied to create titles for unseen documents. Compared to the Rule-based or summarization based approaches, statistical methods rely on the availability of training data, which can be a disadvantage of these approaches. Also, since statistical methods compute the correlation between every news article word and every word in the title throughout the training data, it is computationally more expensive than summarization or Rule-based approaches. These approaches are ill-suited when there is lack of sufficient training data or when computational resources are limited. On the other hand, ability to learn from training data is also its strength and adds robustness to these approaches. Unlike Rule-based or some summarization based approaches in which rules for selecting representative sentences and further pruning them to desired length are built into the system, statistical approaches through model training actually learn how to compose a good title from the training corpus. This ability makes it easier to transport statistical methods to different languages and domains, even making it suitable for cross-lingual title generation tasks. Also in general, statistical approaches are more robust to noise in the articles making them suitable for producing titles from machine generated texts. Also, statistical approaches can be devised to produce titles containing words not restricted to the article.

2.3 Naive Based Approach

Most current statistical approaches are variations of the Naive Bayes approach proposed by Banko, Mittal and Witbrock[4]. In the rest of this report we will refer to it as the BMW approach or BMW model. This was the first work to suggest that within a learning framework

one could divide the title generation task into the two phases of content selection and surface realization (title synthesis). They adopt a Naive Bayes approach in which the system is trained to learn the correlation between a word in the article D and a word in a title. They learn the conditional probability of a word appearing in a title given it appears in the document.

$$P(w \in H | w \in D) = \frac{P(w \in H \wedge w \in D)}{P(w \in D)}$$

According to the above expression, one can simply count how many news articles have word w in their titles and article body and divide it by the number of news articles containing word w in their bodies and use the ratio as the approximation for $P(w \in H | w \in D)$. To enforce the sentence structure and score candidate titles, i.e. compute the probability of a word sequence S ; $P(S)$ they use a bi-gram language model. The overall probability of a candidate summary H consisting of word sequence (w_1, w_2, \dots, w_n) is computed as the product of the likelihood of (i) the terms selected for the summary, (ii) the length of the resulting summary, and (iii) the most likely sequencing of the terms in the content set.

$$P(w_1, w_2, \dots, w_n | D) = \prod_{i=1}^n P(w_i \in H | w_i \in D) \cdot P(\text{len}(H) = n) \cdot \prod_{i=2}^n P(w_i | w_1, \dots, w_{i-1})$$

In the BMW model $P(w \in H | w \in D)$ is actually an approximation for $P(w \in H | D)$. Thus all evidence to infer whether the word w should be added to a title or not is based simply on the occurrence of w in the news article. While computing $P(w \in H | D)$ is infeasible because of the infinitely large sample space of the document D , a better approximation than $P(w \in H | w \in D)$ can be arrived upon by considering not just the word occurrence in the article but instead considering the surrounding context of the word along with the word in the news article. We will see in Chapter 4 that an overlapping feature set consisting of word n-grams, word POS, POS n-grams, word tf.idf measure, word position in text and others provides a good ‘macro-level’ evidence for inference and a better approximation to the content selection model. Another deficiency of the BMW model is that it constrains the choice of title words and does not allow words outside of the article to be used as words in the title. Further, news article words that were never observed in any of the titles in the training data would have a 0 probability of being included in the title for a test news article based on this model. In other words, words present in the titles of the training data are the

only words that could be present in the title of the test data as well, and such a model becomes too restrictive. In this we propose techniques for Content Selection that not only give us a better approximation to $P(w \in H / D)$ but also get rid of the above restriction.

2.4 Rule Based Approaches

While similar to Summarization (Extractive) approaches, techniques in this category create a title for a news story using linguistically motivated heuristics that guide the choice of a potential title. Hedge Trimmer is an example of this category which uses a parse and trim scheme[5]. The system creates a title for a news article by removing constituents from the parse tree of the lead sentence of the article until a certain length threshold is reached. Linguistically motivated techniques guide the choice of what constituents should be removed and retained. The principal advantage of these techniques is that they do not require prior training on a large corpus of title-story pairs since there is no model to be learnt. On the other hand, deciding which single sentence best reflects the contents of the entire news article is a difficult task. Often, news stories have important pieces of information dispersed throughout the article and the approach of trimming the lead or a single important sentence may be unsuccessful in practice. The approach in Hedge Trimmer is very similar to the sentence compression work of Knight and Marcu, where a single sentence is shortened using statistical compression.

3. ~~DOCUMENT CLASSIFICATION~~

It is important to emphasize that getting from a collection of documents to a classification of the collection, is not merely a single operation, but is more a process in multiple stages. These stages include more traditional information retrieval operations such as crawling, indexing, weighting, filtering etc. Some of these other processes are central to the quality and performance of most clustering algorithms, and it is thus necessary to consider these stages together with a given clustering algorithm to harness its true potential. We will give a brief overview of the classification process, before we begin our literature study and analysis. We have divided the offline clustering process into the four stages outlined below. The Stages of the Process of Classification:

Collection of Data includes the processes like crawling, indexing, filtering etc. which are used to collect the documents that needs to be clustered, index them to store and retrieve in a better way, and filter them to remove the extra data, for example, stopwords. Here we have used BBC news dataset.

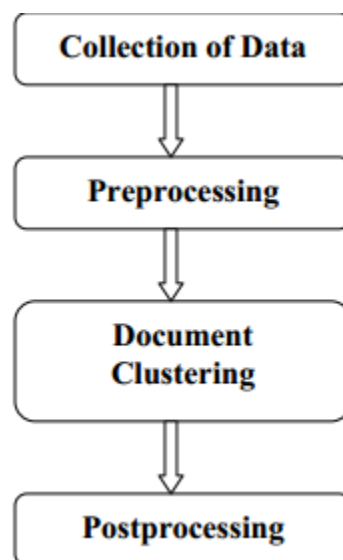


Figure.1

Preprocessing is done to represent the data in a form that can be used for classification. There are many ways of representing the documents like, Vector-Model, graphical model, etc. Many measures are also used for weighing the documents and their similarities.

Stressing simplicity first, our feature vectors were built only from text that would be seen on the screen, i.e. normal document text, image captions and link text, and no extra weight was given according to emphasis (bold typeface, italic typeface, different colours, etc.). For each document, the extraction process was as follows:

- The set of all words that appeared at least once in the document was extracted.
- If stop-word removal was switched on, we removed from the set of extracted words any word that was listed in our stop-word list.
- If word stemming[6] was switched on, we combined all the words with a similar stem, (i.e. count all occurrences of a word as a single occurrence of its stem).

Processing for finding the importance of each word or term we have used TF-IDF(*term frequency-inverse document frequency*).

The TF-IDF weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the TF-IDF weighting scheme are often used.

The importance increases proportionally to the number of times a word appears in the individual document itself--this is called Term Frequency. However, if multiple documents contain the same word many times then you run into a problem. That's why TF-IDF also offsets this value by the frequency of the term in the entire document set, a value called Inverse Document Frequency

$$idf(t, D) = \log \frac{|D|}{1 + |\{d \in D : t \in d\}|}$$

- The numerator: $|D|$ is referring to our document space. It can also be seen as $D = d_1, d_2, \dots, d_n$ where n is the number of documents in your collection. Thus, for our example $|D| = 4$, the size of our document space is 4, since we're only using 4 documents.
- The denominator: $|\{d \in D : t \in d\}|$ implies the total number of times in which term t appeared in all of your document d (the $d \in D$ restricts the document to be in your current document space). Note that this implies it doesn't matter if a term appeared 1

time or 100 times in a document, it will still be counted as 1, since it simply did appear in the document. As for the plus 1, it is there to avoid zero division.

Document Clustering: Now that we have our matrix with the term frequency and the IDF weight, so we first converted them to vector form so that we could provide the input to the SVM classifier. But before converting it in vector form we divided our input dataset into two different categories i.e. training and testing.

Document Classifier

Document Classifier

Please Select Classifier:

Please Select Test Size:

Stop Word Remove?

Classify

Error

Result of Classifier:

Selectd Classifier:

Stop Word Removed:

Accuracy :

Figure.2

The screenshot shows a window titled "Document Classifier". Inside, there's a section titled "Document Classifier" in pink. Below it, there are three input fields: "Please Select Classifier:" with a dropdown menu showing "Support_Vector_Machine(SVM)", "Please Select Test Size:" with a dropdown menu showing "0.5", and "Stop Word Remove?" with a dropdown menu showing "Yes". Below these fields is a blue "Classify" button. Under the button is a yellow "Error" label. Below that is a red "Result of Classifier:" label. At the bottom, there are three rows of output labels: "Selectd Classifier:", "Stop Word Removed:", and "Accuracy :". Each label has a corresponding colored bar to its right: green for "Selectd Classifier:", pink for "Stop Word Removed:", and light green for "Accuracy :".

Figure.3

The screenshot shows the same "Document Classifier" window, but now the "Classify" button is disabled (greyed out). The "Error" label is still yellow. The "Result of Classifier:" label is still red. The output labels at the bottom now have text in their corresponding colored bars: "Selectd Classifier:" has "Support_Vector_Machine (", "Stop Word Removed:" has "Yes", and "Accuracy :" has "0.9694519317160827".

Figure.4

4. TITLE GENERATION

A) Content selection: This step assign each word in the news story a probability of its inclusion in the title.

B) Title synthesis: this step assign a score to the sequence of surface word ordering of a particular title candidate by modeling the probability of title word sequence in the context of the news story.

C) Decoding: In this step, we use a decoding algorithm which explore the space of candidate title hypothesis to generate optimal title word sequence.

4.1 Extracting Keywords

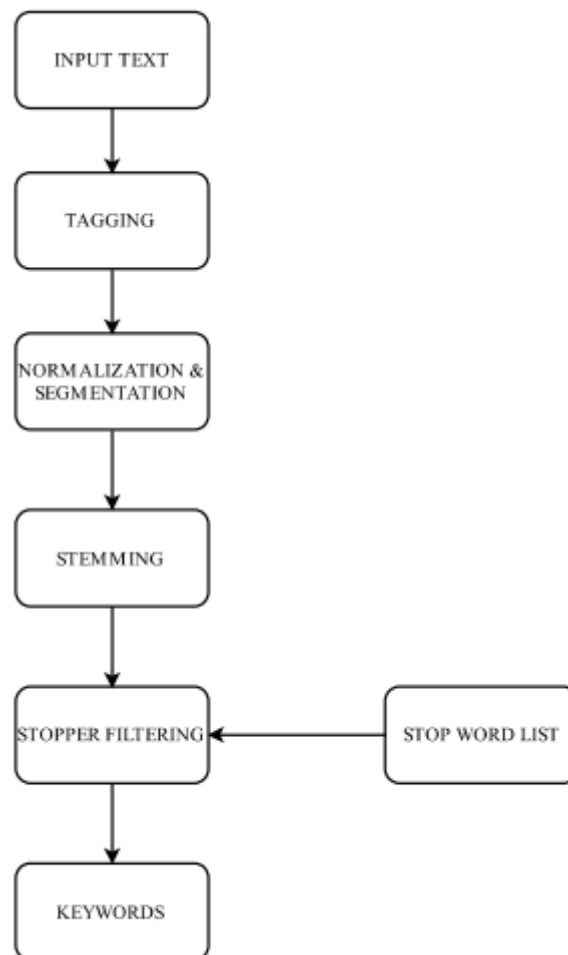


Figure.5

4.1.1 Tagging

In this phase, each of the tokens in the document is annotated with the part of speech

information. **This is done using RDRPOSTagger** – It's a rule based tagger that employs an error-driven approach to automatically construct tagging rules in the form of a binary tree. The sample sentence, when tagged, we have the following kind of information about the words in the sentence.

“For/**IN** the/**DT** second/**JJ** time/**NN** in/**IN** a/**DT** week/**NN** ,/, the/**DT** country's/**NN**
 technology/**NN** hub/**NN** had/**VHD** reasons/**NNS** to/**TO**”

4.1.2 Normalization

Normalization is the process of **removing unnecessary text from the document**. The unnecessary text may include **text found between hyphens, after bullets, and between parentheses**.

4.1.3 Segmentation

This phase outputs the document as a set of words by removing the unnecessary semicolons, colons, exclamation marks etc. The full-stops are retained to indicate end of sentence which is needed during further stages of title generation. After this stage the sample sentence would produce the following set of words:

“For the second time in a week the country’s technology hub had reasons to celebrate as Wipro Ltd today joined the exclusive club of infotech firms with a billion-dollar revenue.”

4.1.4 Stemming

The segmented words are then queried in the WordNet Keywords Extraction Process to get the **root form of the words**. For example words like issues , relinquishing are transformed to *issue*, *relinquish* resp. The part of speech information, already assigned to each word by the tagger, is used here to query the WordNet to get the root form of the word in the sense in which it is used in the document. The output is thus the set of root forms of the words of the original document as shown below for the sample sentence:

“For the second time in a week, the country technology hub have reason to celebrate as Wipro Ltd today join the exclusive club of InfoTech firm with a billion-dollar revenue.”

4.1.5 Stopper Filtering

There is a precompiled set of fluff words. This stoplist consists of a list of common function words such as determiners (a, the, this), prepositions (in, from, to), conjunctions (after, since as), coordination (and, or). Also those words which occurs more frequently but contribute little meaning like *about*, *them*, *only* etc. During this stage the segmented set of words of the document are filtered through this list of fluff words. The remaining words form the content words of the document. The content words for the sample sentence are the following: “time week country technology hub celebrate Wipro Ltd today join exclusive club InfoTech firm billion-dollar revenue.”

4.1.6 Finding Keywords

The content words are then queried in the WordNet thesaurus to find out the synonyms of each word for the particular part of speech annotation with which it is used in the document. The number of important words thus get reduced by merging information about words conveying similar content. The final or condensed set of root form of these content words after this phase represents the set of keywords. In many of the cases the set of keywords are the same as the set of content words obtained from the previous phase as in the case for the sample sentence.

4.2 Maximum Entropy Framework

Many problems in natural language processing (NLP) can be formulated as classification problems, in which the task is to estimate the correct linguistic “outcome” $a \in A$ given some “context” information $b \in B$. This involves constructing a classifier function $cl: B \rightarrow A$, which in turn can be implemented with a conditional probability distribution p , such that $p(a/b)$ is the probability of “class” a given some “context” b . Contexts in NLP tasks can vary from fairly simple (single word) to complex (multi-words and associated labels and tags). Large text corpora usually contain some information about the co-occurrence of a ’s and b ’s, but never enough to reliably estimate $p(a/b)$ for all possible (a, b) pairs, since the contexts in b are typically sparse. The challenge is then to utilize a method for using the partial evidence about the a ’s and b ’s to reliably estimate the probability model p . Maximum entropy (log linear) probability models offer a clean way to combine diverse pieces of contextual evidence in order to estimate the probability of a certain linguistic

outcome occurring with a certain context. We discuss how evidence can be represented in the form of feature functions and explain how to formalize a training problem as a probability model estimation problem under both the Maximum Likelihood framework and the maximum entropy framework. We also discuss that models arrived at by both the methods are essentially the same.

4.2.1 Feature Function

We extract evidence presented by training data with the help of feature functions and contextual predicates, a terminology and notation which has become standard when discussing maximum entropy frameworks in NLP. Let A be the set of possible outcomes $\{a_1, \dots, a_n\}$ and B be the sample space of all possible context information. Then the contextual predicate is a function of the form:

$$cp : B \rightarrow \{true, false\}$$

cp evaluates to true or false, corresponding to presence or absence of some information in the context b . In other words, contextual predicates can be thought of as filter functions. Contextual predicates are designed by the experimenter and used in feature functions of the form:

$$f : A \times B \rightarrow \{0, 1\}$$

Throughout this work, a feature would be represented as:

$$f_{cp, a'}(a, b) = \begin{cases} 1 & \text{if } a = a' \text{ and } cp(b) = true \\ 0 & \text{otherwise} \end{cases}$$

A feature checks for the co-occurrence of some outcome a and contextual predicate cp evaluating to *true*. The actual set of features for a particular problem is decided by the feature selection strategy and is influenced by the problem domain.

4.3 Title Generation Model

In this section, we present our title generation model in detail. In particular, we propose Log-Linear discriminative models for both sentence Content selection and Title synthesis. Log-Linear or alternatively Maximum Entropy models have been successfully applied in the past to important NLP problems such as parsing, Part of Speech (POS) tagging, Machine

Translation, Sentence boundary detection, Ambiguity resolution, etc. This class of models, also known as log-linear, Gibbs, exponential, and multinomial logit models, provide a general-purpose machine learning technique for classification and prediction which has been successfully applied to fields as diverse as computer vision and econometrics. A significant advantage of Maximum Entropy mode is that they offer the flexibility and ability to include a rich and complex feature set spanning syntactic, lexical and semantic features. They also offer a clean way to incorporate various information sources and evidences into a single powerful model. Depicts the overall framework of our title generation system. Given a news story, for which k -best titles are to be generated, the core of the system uses maximum entropy models for both content selection and title synthesis and a decoding algorithm that explores the space of candidate title hypotheses to generate the optimal title word sequences. The decoding algorithm uses the title synthesis model $PW S$ to score candidate title sequences. The title synthesis model uses the content selection scores for the words in the sequence as one of the feature functions within the model. The content selection model assigns each word in the news story a probability of its inclusion in the title. Additionally the title synthesis model uses four other feature functions over the sequence of words in the form of a language model (PLM), a POS language model ($PP OS LM$), a title length feature ($PLEN$) and a n-gram match model ($PMATCH$). The decoding algorithm is preceded by a pre-processing phase in which the news story undergoes tokenization, removal of special characters (cleaning) and part of speech tagging. The optimal title sequences can optionally undergo some form of post processing. For example, verbs in the title can undergo morphological variation since title verbs are typically in present tense.

4.3.1 Content Selection Model

Content selection requires the system to learn a model of the relationship between the appearance of some features in a document and the appearance of corresponding features in the title. The simplest way of modelling this relationship is to estimate the likelihood of some token appearing in a title given that the token (or possibly a set of tokens) appears in the document to be summarized. Put simply, content selection assigns each document word the probability of being included in the title. At the same time, it should be noted that it is not Content Selection alone, but both Content Selection and Title Synthesis (Ordering or Realization) that influence whether a word is included in the title. Ideally content selection should be modelled as $P(w \in H / D)$ i.e. the probability of word inclusion in the title given the whole document. But since the sample space of

documents or news stories can be infinitely large it is infeasible to learn such a model. Hence various content selection strategies try to approximate computation of $P(w \in H / D)$. One such approximation is the Naive Bayes content selection model.

$$P(w \in H|D) \approx P(w \in H|w \in D) = \frac{P(w \in H \wedge w \in D)}{P(w \in D)}$$

The model can be very easily estimated by counting the number of news articles having word w in their titles and article body and divide it by the number of news articles containing word w in their bodies. But a better approximation than $P(w \in H / w \in D)$ to content selection can be arrived upon by also considering the context surrounding the word and the word's positional and domain relevant importance (TF*IDF measure) information, since such a model combined evidences from multiple sources. Formally, for content selection, $pCS(yw / cx(w))$ denotes the probability of including the word w in the title, given some contextual information $cx(w)$. $1 - pCS(yw / cx(w))$ is the probability of not including in the title. Our goal is to build a statistical model $pCS(yw / cx(w))$ for content selection that best accounts for the given training data, i.e. a model p which is as close as possible to the empirical distribution p_e observed in the training data. A Conditional Maximum Entropy (log linear) model for content selection has the following parametric form.

$$pCS(yw|cx(w)) = \frac{1}{Z(yw)} \exp\left[\sum_{i=1}^k \lambda_i f_i(yw, cx(w))\right]$$

where, $f_i(yw, cx(w))$ are binary valued feature functions that map some form of relationship between the word w and its context $cx(w)$ to either a 0 or 1. λ denotes weights for feature functions. λ_i is the weight for the feature function f_i . The greater the weight, the greater is the feature's contribution to the overall inclusion or non-inclusion probability. k is the number of features and $Z(yw)$ is the Normalization constant to ensure probability of all outcomes (inclusion and non-inclusion of w in title) sums to 1. Given the training data, there are numerous ways to choose a model p that accounts for the data. It can be shown that the probability distribution of the form is the one that is closest to p_e in the sense of minimizing the Kullback –Leibler (KL) divergence between p_e and p , when subjected to a set of feature constraints. The Principle of Maximum Entropy is based on the premise that when estimating the probability distribution, one should select that distribution which leaves you with the largest remaining uncertainty (i.e., the maximum entropy) consistent with the given empirically observed counts.

$$P = \{p | E_p f_i = E_{\tilde{p}} f_i, i = \{1, \dots, k\}\}$$

4.3.2 Features set for Content Selection

The likelihood of a story word being included in the title depends on the feature vector \mathbf{f} and corresponding feature weights \mathbf{w} . Here we present the feature representation $f_{cp,yw}(yw, cx(w))$ over the word w and its surrounding context $cx(w)$. yw indicates the inclusion or non-inclusion of w in the title for values of 1 and 0 respectively. cp is the contextual predicate which maps the pair $\langle yw, cx(w) \rangle$ into true or false. Mathematically the feature function can be represented as below.

$$f_{cp,yw}(yw, cx(w)) = \begin{cases} 1 & \text{if } yw = y'_w (0/1) \text{ and } cp(x) = \text{true} \\ 0 & \text{otherwise} \end{cases}$$

We say that the feature *triggers* if the feature function evaluates to 1, else we say the feature function fails. The word context we consider consists of two words to the left and two words to the right of the word under consideration. The context also includes Part-of Speech (POS) Tags of all five words, word position information of the current word and the TF*IDF score of the current word. To help understand the subsequent discussion on the feature set consider the following extract from a news story as the example.

"... As/RB much/RB as/IN we/PRP might/MD
[try/VB to/TO protect/VB systems/NNS with/IN]
new/JJ systems/NNS "

The word under consideration is *protect* and it has a POS tag VB. Also during training, since the stories are accompanied by actual titles the value of yw can be determined to be either 0 or 1. To help our discussion let us assume that *protect* is present in the corresponding title indicating that $y_{protect}$ has a value 1.

4.3.3 Word/Part-of-speech features

The first set of features are over adjacent words in the context. These include the current literal token (word), word bi-grams, part-of-speech (POS) bi-grams, the part-of-speech (POS) tri-grams and the POS of each word individually. We consider the word and POS n-gram features both in the forward and backward direction from the word under consideration. These features are meant to indicate likely words to include in the title as well as provide

some level of grammaticality. The features are explained with examples below. Among other information in the context, assume that 1) “protect” is not present in the lead sentence, 2) is present in the top 10% of the news story, 3) has its first occurrence in the top 10% of the news story and 4) has a TF-IDF score in the range of top 10 – 20.

- **Current Story Word:** This feature triggers if the current word in the feature matches the word in the contextual predicate *cp* and the corresponding outcomes match as well.
- **Word Bi-gram Context:** This feature triggers if the word bi-gram in the feature matches the word bi-gram in the contextual predicate *cp* and the corresponding outcomes match.
- **POS of Current Story Word:** This feature triggers if the POS tag of current story word equals the POS tag in the contextual predicate *cp* and the corresponding outcomes match. POS tags that are most likely to be included in the title would get higher weights relative to other POS tags, at the end of model training.
- **POS Bi-gram of Current Word:** This feature triggers if the POS tag pair of the current story word and previous (next) word equals the POS tag pair in the contextual\ predicate and the corresponding outcomes match. The intuition is that POS tag pairs that are more common in the title would get higher weight as a feature at the end of training.
- **POS Tri-gram of Current Word:** This feature triggers if the POS tag tuple of current story word and previous (next) word and its previous (next) word equals the POS tag tuple in the contextual predicate and the corresponding outcomes match. Along with POS bi grams this feature encodes some level of grammaticality in the content selection model. This set of features is meant to encode lexical tokens and POS contexts that are commonly seen (included) in titles. However, it should be pointed out that they do so without a larger picture of the function of each word in the sentence. For instance, whether the current word is part of a Noun clause or a Verb clause is not encoded in the context information. Excluding frequently occurring main verbs in the title is uncommon, since that verb and its arguments typically encode most of the information being conveyed. However words within a relative clause for instance may be dropped from the title.

4.3.4 Positional Features

The second set of features are based on the positional information of the word in the news story. Experiments and empirical studies have found that a significant proportion of title words are chosen from the first (lead) sentence of the news story. Similarly in many news stories concluding sentences also convey vital information. Hence the position (in terms of word distance) relative to the beginning of the news story, provides an important cue for its

inclusion or non inclusion in the title. For this we consider the following 3 positional features.

- **Word Positioning Lead sentence:** This features triggers if the word under consideration is present in the lead sentence of the story and the corresponding outcomes match.

- **Word Position:** Additionally, we also divide the news story into the following three intervals based on word count from the beginning.

- $\leq 10\%$ - words occurring in the top 10% of the news story

- $\geq 90\%$ - words occurring in the last 10% of the news story

- $10 < 90\%$ - words occurring in the remainder of the news story not covered by the previous two ranges. This feature triggers if the current word w and the range matches that of the contextual predicate and the corresponding outcomes match.

- **First Word Occurrence Position:** Many title words, mostly proper nouns frequently repeat throughout the news story. Along with the frequency of occurrence, the position of the first occurrence of the word in the news story provides an important cue. The intervals for this feature are similar to the “Word Position” feature above. This feature triggers if the current word w and the range of first word occurrence matches that of the contextual predicate and the corresponding outcomes match.

Word Frequency Based Features: This set of features is based on the frequency of occurrence of a word in the document and its frequency of occurrence in the corpus as a whole. Title words (particularly nouns) tend to occur frequently throughout the news story, at the same time we have to avoid interpreting common words (stop words) such as articles: 'the', 'an', 'a', pronouns: 'this', 'that', etc and prepositions: 'from', 'to', etc as being important based on word frequency. The $tf * idf$ statistical measure as discussed below is precisely the metric to interpret word frequencies in such a manner.

- **Word TF. IDF Range:** This is a novel addition to our feature set and is motivated by information retrieval heuristics. The $TF.IDF$ weight (term frequency inverse document frequency) is a measure often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the $TF.IDF$

weighting scheme are often used by search engines to score and rank a document's relevance given a user query. A high *TF.IDF* score is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents. We divide the words in the news story into disjoint intervals based on their *TF.IDF* measure. Based on this division we can say whether the current word has a top 10% *TF.IDF* measure, 10 – 20% *TF.IDF* measure and so on. We include a feature function to enable words with high *TF.IDF* scores to be present in the title.

The term frequency in the given document is simply the number of times a given term appears in that document. This count is usually normalized to prevent a bias towards longer documents (which may have a higher term frequency regardless of the actual importance of that term in the document) to give a measure of the importance of the term t_i within the particular document.

$$tf(t_i) = \frac{f_i}{\sum_k f_k}$$

Secondly, assume there are N documents in the collection, and that term t_i occurs in n_i of them, then the *inverse document frequency* measure of term t_i is given by

$$idf(t_i) = \log \frac{N}{n_i}$$

Finally,

$$TF.IDF(t_i) = tf(t_i) * idf(t_i)$$

• **Stop Word Feature:** As mentioned earlier, due to high frequency of stop words in the news story, they can be deemed as important title words when actually they are not. So we also include a feature that fires when the considered word is a stop word. Below is a complete list of stop words we consider in this feature.

'a', 'about', 'an', 'are', 'as', 'at', 'be', 'by', 'com',
 'for', 'I', 'from', 'how', 'in', 'is', 'it',
 'of', 'on', 'or', 'that', 'the', 'this', 'to', 'was',
 'when', 'where', 'who', 'will', 'with', 'the', 'www'

4.4 Word Translational Model

Consider the news snippet and corresponding title:

Title: Latin states SET TO BACK Panama for U.N. seat

Story: “Latin American and Caribbean nations on Thursday HEADED TOWARD ENDORSING Panama for an open U.N. Security Council seat after a divisive battle between U.S.-supported Guatemala and Venezuela.”

News Story Example showing title word translation. The example highlights a very common feature of how titles are constructed from news stories by changing the verb word forms. In the above example, *headed toward* is reproduced in the title as *set to* and *endorsing* as *back*. This reproduction usually happens for words which have a verb POS form and is not typically done for nouns, adjectives and other word forms. Also in the case of titles, verbs and often action verbs such as *killed*, *destroyed*, etc are very important to essence of the title. In this case *killed* and *destroyed* could have as well occurred as *shot* and *damaged*. This motivates us to include a word translation probability model for words with verb form so that we have the added flexibility of generating title words out of the current news story. We define the word translation probability model as:

$$p_{WT}(w_i \in H | w_j \in D) = \frac{p(w_i \in H, w_j \in D)}{p(w_j \in D)}$$

Where, H is the title and D is the news story. The word translation probability p_{WT} in equation can be estimated by counting the number of times w_i occurs in title and w_j occurs in news story in the training data and dividing it with the number of times w_j is observed in the news stories in the training data. This is the maximum likelihood estimation of the word translation probability. Thus for instance if $p_{WT}(killed \in H | shot \in D)$ has a high probability relative to other possible substitutions for “shot” then we can substitute “shot” with “killed” in the title with high confidence. Now let W_{vb} be the set of all words that have verb forms and that occurred in any of the titles in the training data. Combining the Content selection model and Word Translation model, for any $w_i \in W_{vb}$ we could write,

$$p_{CS}(y_{w_i} \in H | w_j \in D) = p_{WT}(w_i \in H | w_j \in D) p_{CS}(y_{w_j} \in H | w_j \in D)$$

In other words, we just multiply the content selection probability of word w_j with the probabilistic weight with which another word w_i can be substituted for it. As we will see later on this model allows us to expand word choices for the content selection model beyond the default bag of words present in the news story. The above equation for calculating content selection probabilities of substituted words is based on the assumption that content selection of word w_i is dependent only on w_j 's word translation model and is independent of contents of the news story or the actual content selection model for the news story. The following toy example explains the use of word translation probability. Consider the content selection probability of a particular instance of the word "nations" in a news story D to be 0.30, i.e. $PCS(y \text{ "nations" } / D) = 0.30$.

4.5 Title Synthesis Model

In our framework for title generation, the process of title generation is divided into two phases, the phase of finding good title words for a news story and the phase of organizing selected title words into sequences. While the first conditional maximum entropy model takes care of the first phase: Content Selection, the second conditional maximum entropy model takes care of the second phase: Title synthesis. Title Synthesis or alternatively Surface Realization assigns a score to the sequence of surface word ordering of a particular title candidate by modelling the probability of title word sequences in the context of the news story. Uses the simplest form of word ordering model in the form of a bi-gram language model to reasonable effect. The probability of a word sequence is approximated by the product of the probabilities of seeing each term given its immediate left context. A major drawback of using a bi-gram language model for scoring title candidates is that it fails to consider any context provided by the news story as the scoring is done on a word sequence independent of the contents of the news story. Our title scoring function is motivated by the use of Maximum Entropy models in Statistical Machine translation proposed by Och and Ney in which the best performing statistical models combine different models or knowledge sources (translation model, language model, alignment model, etc.) using maximum entropy parameter estimation. The popular source-channel approach in Machine Translation is contained as a special case in the Maximum Entropy Model. In this framework, for example given a French source language sentence $f1J$ and a candidate English translation $e1I$, the machine translation probability is given by

$$Pr(e_1^I | f_1^J) = p_{\lambda_1^M}(e_1^I | f_1^J) = \frac{\exp[\sum_{m=1}^M \lambda_m h_m(e_1^I, f_1^J)]}{\sum_{e_1^I} \exp[\sum_{m=1}^M \lambda_m h_m(e_1^I, f_1^J)]}$$

In this framework, there are a set of M feature functions and for each feature there exists a model parameter λ_m , $m = 1, 2, \dots, M$ for each feature. We extend a similar idea to our title generation framework. Our title scoring function takes into consideration 5 main aspects (or information sources) of the generated sequence of words. These information sources are contained within a conditional maximum entropy model over the sequence of words that evaluates the candidate title sentences and assigns each candidate a score. Given a sequence of words $H = w_1, w_2, \dots, w_n$ as a candidate title for a news story D , the Whole Sentence title synthesis model is given by

$$p_{WS}^{\alpha_M}(H = w_1, w_2, \dots, w_n | D) = \frac{\exp[\sum_{m=1}^M \alpha_m h_m(H = w_1, w_2, \dots, w_n)]}{\sum_{H'} \exp[\sum_{m=1}^M \alpha_m h_m(H' = w'_1, w'_2, \dots, w'_n)]}$$

In this model, there are $M = 5$ feature functions and for each feature there exists a model parameter α_m , $m = 1, 2, \dots, M$. The 5 feature functions (information sources) in this maximum entropy model.

- **Language Model Feature:** A bi-gram language model trained on a training corpus consisting only of title sentences is used to assign the sequence of words in the candidate title a score as below.

$$h_1(H = w_1, w_2, \dots, w_n) = h_{LM}(H = w_1, w_2, \dots, w_n) = \sum_{i=2}^n \log(P(w_i | w_{i-1}))$$

- **POS Language Model Feature:** A Part-of-Speech tri-gram language model trained on a Part-of-Speech annotated training corpus consisting only of title sentences is used to assign the sequence of Part-of-Speech of words in the candidate title a score as below.

$$\begin{aligned} h_2(H = w_1, w_2, \dots, w_n) &= h_{POS_LM}(H = w_1, w_2, \dots, w_n) \\ &= \sum_{i=3}^n \log(P(POS_{w_i} | POS_{w_{i-1}}, POS_{w_{i-2}})) \end{aligned}$$

- **Title Length Feature:** A title length probability distribution is computed

$$h_3(H = w_1, w_2, \dots, w_n) = h_{LEN}(H = w_1, w_2, \dots, w_n) = \log(P(len(H) = n))$$

on the training corpus consisting of title sentences to assign the length of candidate title sentence a probability. This feature biases the length of generated candidate titles to be within typical observed lengths of 5 to 15 words and penalizes titles with too short or too long a length.

- **Content Selection Feature:** Content selection scores that were computed for each story word during the content selection phase are also critical information sources. Words with high CS scores would bias the title synthesis model into choosing sequences with words having high content selection probabilities. The content selection feature score is the sum of log probabilities of content selection probabilities of each word in the sequence.

- **N gram Match Feature:** In order that the title synthesis model does not choose words from disparate sections and sentences of the news story and to ensure that word sequences in the title maintain some continuity with respect to the news story, we have added a word N-gram match feature where the value of N is 3. N-gram match feature is essentially calculating the logarithm of the BLEU score of the title with respect to the news story instead of reference title sentences. Also, there is no brevity penalty in this case of calculating N gram title match with respect to the news story.

$$h_5(H = w_1, w_2, \dots, w_n) = h_{MATCH}(H = w_1, w_2, \dots, w_n) = \log \sum_{n=1}^N \gamma_n \log(p_n)$$

Where, γ_n are positive weights summing to one. In our system we use, $N = 3$ and uniform weights $\gamma_n = 1/N$. P_n is the N-gram precision using n-grams up to length N, where precision is calculated with respect to the news story. Language model features 1 and 2 enforce grammaticality over the sequence of words, feature 3 penalizes candidate titles straying away from typical title lengths (usually between 5 and 15), feature 4 ensures accuracy of words included in the title through the content selection model, while feature 5 in conjunction with feature 4 enforces coherence and continuity in the generated candidate titles, since contiguous words sequences (word phrases) would be encouraged during selection.

4.6 Decoding Algorithm

As mentioned before, inclusion of a particular word in the title is influenced by both the models: content selection and title synthesis. The decoding algorithm incrementally builds

sequences from left to right in the form of candidate title hypothesis and also systematically combines the content selection model and the title synthesis model together. To find the optimal candidates (paths) we use a Beam search algorithm with pruning. Beam search algorithms have also been widely adopted in Statistical Machine Translation (SMT) systems. The search is performed by building partial sequences (hypotheses), which are stored in a priority list (or lists). Word sequences are scored based on the *whole sentence title synthesis model* in equation. Content selection probabilities are indirectly fed as scores into this model. The priority list helps retain only promising sequences with very high scores and discards all other hypotheses with low scores to make the search feasible. Our decoding or title search algorithm considers as input a news article with the word sequence w_1, w_2, \dots, w_L and the word POS sequence p_1, p_2, \dots, p_L , where L is the length of news story. T , the number of top scoring titles returned by the algorithm is user input. Other inputs to the algorithm are the Content Selection model (parameters): λ , the title synthesis model parameters: α , the word translation model: PWT and the individual model components of the title synthesis model: PLM , $PPOS LM$, $PLEN$. TF-IDF scores and N-gram match scores are computed on the fly, while the IDF component of TF-IDF is computed beforehand on the training corpus. The titles are ranked by the *Whole-Sentence Title Scoring Function* using equation at the end of each iteration and pruned down to maximum of C candidates, in other words C is the hypotheses cut-off. One problem when building a beam search decoder is that decoders tend to bias the search towards those sequences that had higher probabilities during the first stages (initial iterations). This is not always the best scenario, and in order to ensure that all early hypotheses receive fair comparison and compete to stay alive after pruning, we maintain two separate cut offs; C_1 and C_2 . C_1 is used during the initial iterations (0-5) and C_2 is used during subsequent 37 iterations ($6 - ML$). ML is the maximum length of title word sequences and is the last iteration. The values for both C_1 and C_2 were tuned to 20 and 10 respectively in our system. While ML is set to 15 as is typical of title lengths. Alternative approaches to pruning can also be used, such as keeping all hypotheses after increments which have scores lying within a certain radius of the score of the original hypothesis before the increments are done. Additionally, to account for word substitutions due to word translation model we initialize W_{vb} to be the set of all words that have verb forms and which occur in any of the titles in the training data. W_{vb} is further pruned down during the content selection phase in presence of the word translation model using Algorithm 2. The complete algorithm is given in Algorithm 1

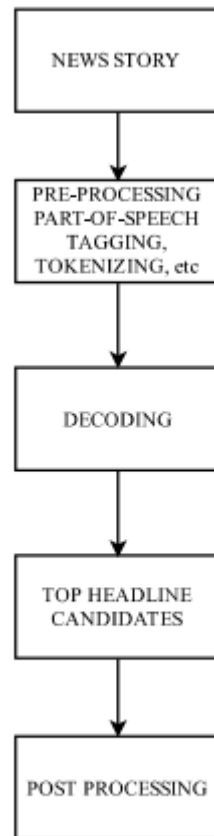


Figure.6

4.7 BLEU

BLEU is a system for automatic evaluation of machine translation that uses a modified n-gram precision measure to compare machine translations to reference human translations. This automatic metric counts the number of n-grams in the candidate that occur in any of the reference summaries and divides by the number of n-grams in the candidate. The size of the n-grams used by BLEU is also configurable. BLEU-n uses 1-grams through n-grams. In case of evaluation of title generation systems, titles or short summaries are treated as a type of translation from a verbose language to a concise one, and compare automatically generated titles to a set of human generated titles. This treatment of title generation as statistical machine translation is indeed the case for Naive Bayes (BMW) Model and Zajic, et al HMM-Hedge model, both of which treat title generation as a variant of statistical machine translation. Specifically, to evaluate the BLEU score for a set of K candidate translations $h * K$ against a set of references rK , we accumulate n-gram precision

and closest reference length information for each h_k from h_K and compute the logarithm of BLEU score as follows:

$$\log BLEU(h_k^*, r_K) = \left\{ \sum_{g=1}^N w_g \log(p_g) - \max\left(\frac{L_{ref}^*}{L_{sys}} - 1, 0\right) \right\}$$

where, p_n is the modified n-gram precision which counts the number of n-grams matched between the candidate being evaluated and the reference set and divides this count by the number of n-grams in the candidate. $w_n = 1/N$ where N is the n-gram size we want to consider, typically 3. The second term in the equation is also called the brevity penalty. L_{ref}^* is the effective length of reference titles and L_{sys} is the effective length of closest reference length matches for the title candidates.

4.8 RESULTS

Classification



Figure.7

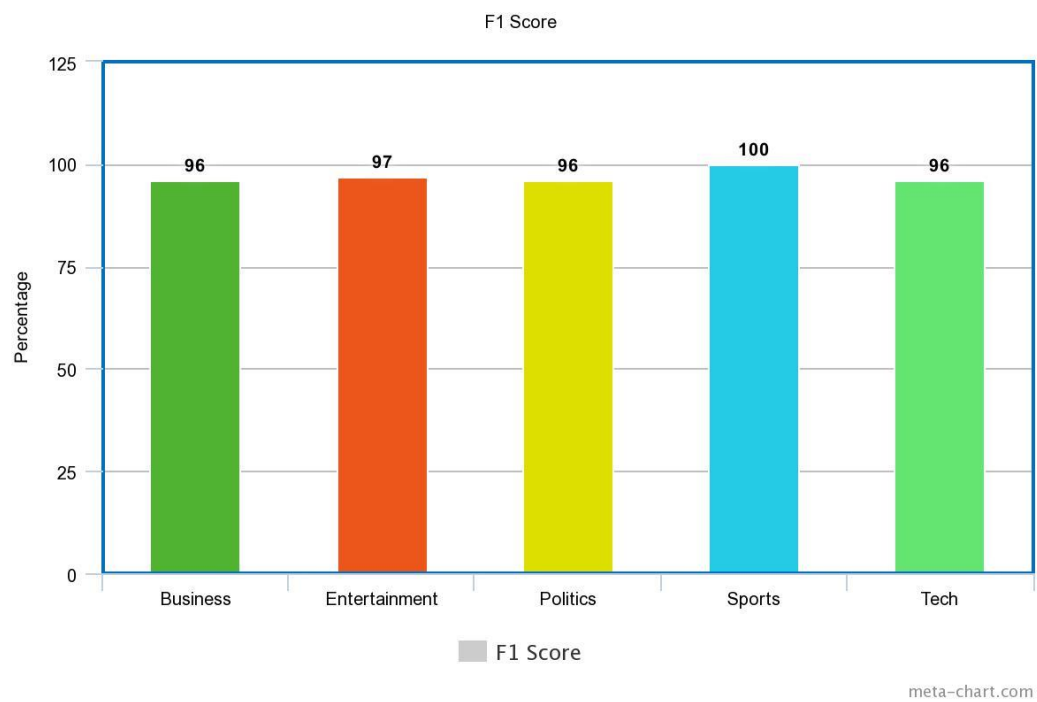


Figure.8

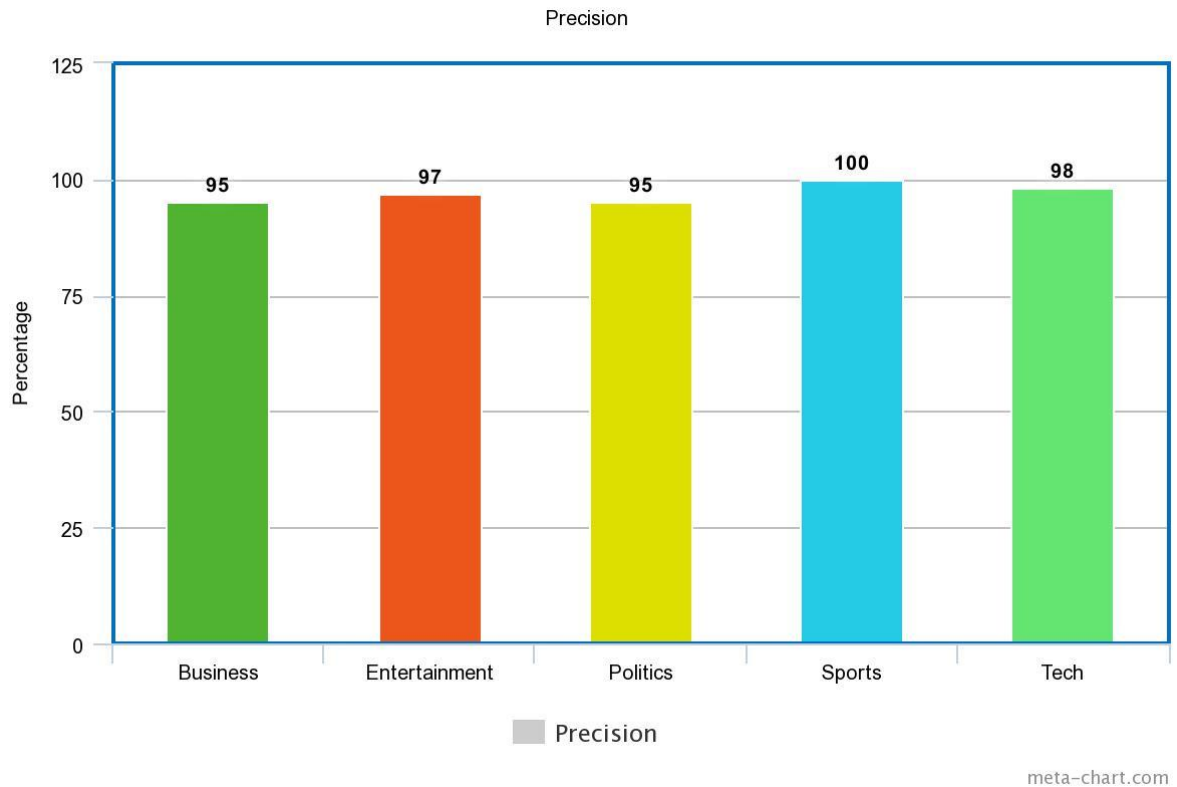


Figure.9

Title Generation

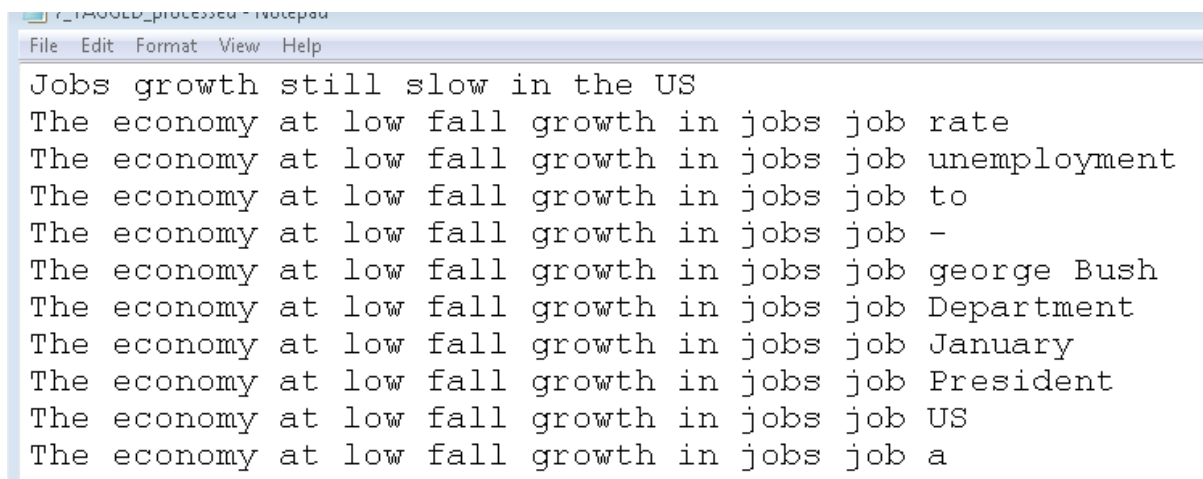
```

1 Jobs growth still slow in the US
2
3 The US created fewer jobs than expected in January but a fall in jobseekers pushed the unemployment rate to its lowest level in thr
4
5 According to Labor Department figures US firms added only 146000 jobs in January.
6 The gain in non-farm payrolls was below market expectations of 190000 new jobs.
7 Nevertheless it was enough to push down the unemployment rate to 5.2% its lowest level since September 2001.
8 The job gains mean that President Bush can celebrate - albeit by a very fine margin - a net growth in jobs in the US economy in his
9 He presided over a net fall in jobs up to last November's Presidential election - the first President to do so since Herbert Hoover
10 As a result job creation became a key issue in last year's election.
11 However when adding December and January's figures the administration's first term jobs record ended in positive territory.
12
13 The Labor Department also said it had revised down the jobs gains in December 2004 from 157000 to 133000.
14
15 Analysts said the growth in new jobs was not as strong as could be expected given the favourable economic conditions.
16 "It suggests that employment is continuing to expand at a moderate pace" said Rick Egelton deputy chief economist at BMO Financial
17 "We are not getting the boost to employment that we would have got given the low value of the dollar and the still relatively low i
18 "That means there are a limited number of new opportunities for workers."

```

Figure.10

Generated top 10 title for above news story with first as original title given by human.



```

7_TAGGED_processed - Notepad
File Edit Format View Help
Jobs growth still slow in the US
The economy at low fall growth in jobs job rate
The economy at low fall growth in jobs job unemployment
The economy at low fall growth in jobs job to
The economy at low fall growth in jobs job -
The economy at low fall growth in jobs job george Bush
The economy at low fall growth in jobs job Department
The economy at low fall growth in jobs job January
The economy at low fall growth in jobs job President
The economy at low fall growth in jobs job US
The economy at low fall growth in jobs job a
  
```

Figurer.11

Title Evaluation:

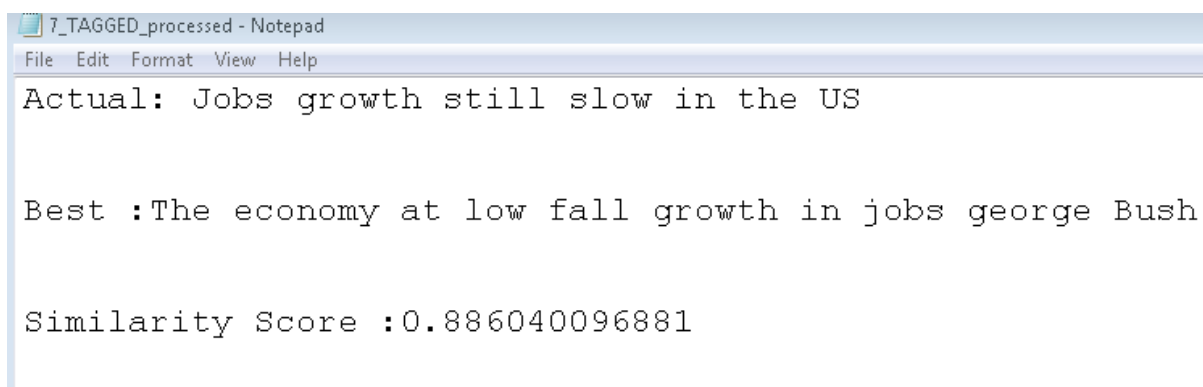
For evaluating, we will use a combination of cosine similarity and word order similarity to compare the generated title with actual title. This will cover both the semantic and syntactic information of the sentences. Similarity between two sentences T1 and T2 will be calculated as:

$$S(T1, T2) = \delta S(T1, T2) + (1 - \delta) R(T1, T2)$$

Where SR and SS is word order similarity and cosine similarity is given by respectively:

$$R = 1 - [(r1 - r2) / (r1 + r2)]$$

$$S = (s1 * s2) / (|s1| * |s2|)$$



```

7_TAGGED_processed - Notepad
File Edit Format View Help
Actual: Jobs growth still slow in the US

Best :The economy at low fall growth in jobs george Bush

Similarity Score :0.886040096881
  
```

Figure. 12

5. COMPLETE CONTRIBUTARY SOURCE CODE

CLASSIFICATION

Training

```
# this is Support vector machine classifier

from sklearn import svm

def train_machine(stories_train, labels_train):

    # three parameter is used of SVM

    # 1. C

    # 2. gamma

    # 3. kernal

    clf = svm.SVC(C=1000000.0, kernel='rbf')

    # fit data to train classifier

    clf.fit(stories_train, labels_train)

    # return trained classifier

    return clf
```

SVM

```
from processing import prediction

from processing import training

from processing import vector

from classification.processing import split_train_test

def perform(stopword, size):

    # get input data vector

    stories_train , labels_train = vector.make_vector(stopword)

    # get split into traninig and testing

    feature_train, feature_test, labels_train, labels_test = split_train_test.split_data(stories_train, labels_train, size)

    #Now train machine

    classifier = training.train_machine(feature_train, labels_train)

    #now make prediction

    pred = prediction.prediction_fun(classifier, feature_test)

    #now get accuracy

    accuracy = prediction.accuracy_score(pred, labels_test)
```

```
#return accuracy
```

```
return accuracy
```

Split_train_test

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.feature_selection import SelectPercentile, f_classif
```

```
def split_data(stories,labels,test_size):
```

```
    # Create the training-test split of the data
```

```
    stories_train, stories_test, labels_train, labels_test = train_test_split(stories, labels, test_size=float(test_size),
random_state=42)
```

```
    # stores_train = training data for classifier
```

```
    # stories_test = testing data for classifier
```

```
    # labels_train = training data for classifier
```

```
    # labels_test = testing data for classifier
```

```
    vectorizer = TfidfVectorizer(min_df=1)
```

```
    stories_train_trans = vectorizer.fit_transform(stories_train)
```

```
    stories_test_trans = vectorizer.transform(stories_test)
```

```
    selector = SelectPercentile(f_classif, percentile=10)
```

```
    selector.fit(stories_train_trans, labels_train)
```

```
    stories_train_transformed = selector.transform(stories_train_trans).toarray()
```

```
    stories_test_transformed = selector.transform(stories_test_trans).toarray()
```

```
    return stories_train_transformed,stories_test_transformed,labels_train,labels_test
```

Prediction

```
#predict the result with testing data
```

```
from sklearn.metrics import confusion_matrix,accuracy_score
```

```
def prediction_fun(clf,stories_test):
```

```
    # test classiefir with argument
```

```
    pred = clf.predict(stories_test)
```

```
    return pred
```

```
def get_accuracy(pred,labels_test):
```

```

accuracy = accuracy_score(pred,labels_test)

return accuracy

```

Vector

```

# This file access each document in each category and
# make a preprocessed text file which contain all the term each document have.
import pandas as pd

def make_vector(needStopword):

    labels = []
    stories = []

    if needStopword=='Yes':
        file_name = 'data/input_withStop.xlsx'
    else:
        file_name = 'data/input_withoutStop.xlsx'

    news_data_frame = pd.read_excel(io=file_name, sheetname='sheet 1')

    labels = news_data_frame['Label'].tolist()
    stories = news_data_frame['News'].tolist()

    return stories, labels

```

TITLE GENERATION

File_Level_Features

```

# -*- coding: utf-8 -*-

# define all the feature functions here

import math

```

```

def cal_totalwords(lines):

    word_list = []

    dict_probs = {}

    for line in lines:

        val = line.strip()

```



```

wordtags = val.split() #word-pos-tag

for entry in wordtags:
    parts= entry.rsplit('/',1)
    if len(parts)>=2:
        word, tag = parts[0],parts[1]
        word_list.append(word)

total_len=len(word_list)
first_range=math.floor((0.1)*total_len)
second_range=math.floor((0.9)*total_len)

dict_probs['total_len'] = total_len
dict_probs['first_range'] = first_range
dict_probs['second_range'] = second_range
return dict_probs

# calculate the word range for the file this ca

def get_word_range(lines):
    """
    Return the dict contain the lead postion of the words in the file i.e if the word is present in the first sentence or
    not
    """
    position_parameters = { }
    total_length = cal_totalwords(lines)
    # if the first sentence is encountered add lead postion value to one for that particular word
    firstSentence = 1
    current_count = 0

    for line in lines:
        line = line.strip()
        if firstSentence==1:
            wordtags = line.split() # word-pos-tag
            for entry in wordtags:
                current_count += 1
                parts = entry.rsplit('/',1)

```

```

if len(parts)>=2:
    word, tag = parts[0],parts[1]
    if word not in position_parameters:
        position_parameters[word] = { }
        position_parameters[word]['range'] = []
        position_parameters[word]['lead_sentence'] = 1
        position_parameters[word]['first_occurance'] = current_count
    # Added the word postion for each token and also added the word rage for given word

    if current_count <= total_length['first_range']:
        position_parameters[word]['range'].append('1_10')
    elif total_length['total_len'] >= current_count >= total_length['second_range']:
        position_parameters[word]['range'].append('90_100')
    else:
        position_parameters[word]['range'].append('10_90')

    #returns the word position for each word,word lindex is associated with current count
firstSentence=0
else:
    wordtags= line.split()
    for entry in wordtags:
        current_count += 1
        parts = entry.rsplit('/', 1)
        if len(parts)>=2:
            word, tag = parts[0],parts[1]
            if word not in position_parameters:
                position_parameters[word] = { }
                position_parameters[word]['range'] = []
                position_parameters[word]['lead_sentence'] = 0
                position_parameters[word]['first_occurance'] = current_count
            # Added the word postion for each token and also added the word rage for given word
            if current_count<=total_length['first_range']:
                position_parameters[word]['range'].append('1_10')

```

```

elif total_length['total_len'] >= current_count >= total_length['second_range']:

    position_parameters[word]['range'].append('90_100')

else:

    position_parameters[word]['range'].append('10_90')

return position_parameters

```

Features

define all the feature functions here

"""List of features used:

1. Current Story Word
2. Word Bi-gram Context - both sides -1 and +1
3. POS of Current Story Word
4. POS Bi-gram of Current Word - both sides -1 and +1
5. POS Tri-gram of Current Word - both sides -1, -2 and +1, +2
6. Word Position in Lead sentence
7. Word Position
8. First Word Occurrence Position
9. Word TF-IDF Range

"""

def get_outcome(word, heading):

"""Returns if the word is present in heading or not.

"""

word = word.rsplit('/', 1)[0]

if word in heading:

return 1

return 0

def get_feature_dict(word_tag_list, index=2):

"""

Returns the feature dictionary of POS tagged list of words(5 words are passed in the list).

"""

dict_feature = {}

word_list = []

```

tag_list = []

for entry in word_tag_list:
    word, tag = entry.rsplit('/', 1)
    word_list.append(word)
    tag_list.append(tag)

# 1. Current Story Word
# 3. POS of Current Story Word
dict_feature['1_w'] = word_list[index]
dict_feature['3_t'] = tag_list[index]

# 2. Word Bi-gram Context - both sides -1 and +1
# 4. POS Bi-gram of Current Word - both sides -1 and +1
# 5. POS Tri-gram of Current Word - both sides -1, -2 and +1, +2
# 6. Word Position in Lead sentence
# 7. Word Position
# 8. First Word Occurrence Position

if index-1 >= 0:
    dict_feature['2_w_w-1'] = '%s,%s' %(word_list[index], word_list[index-1])
    dict_feature['4_t_t-1'] = '%s,%s' %(tag_list[index], tag_list[index-1])
if index-2 >= 0:
    dict_feature['5_t_t-1_t-2'] = '%s,%s,%s' %(tag_list[index], tag_list[index-1], tag_list[index-2])

if index < len(word_list)-1:
    dict_feature['6_w_w+1'] = '%s,%s' %(word_list[index], word_list[index+1])
    dict_feature['7_t_t+1'] = '%s,%s' %(tag_list[index], tag_list[index+1])

if index < len(word_list)-2:
    dict_feature['8_t_t+1_t+2'] = '%s,%s,%s' %(tag_list[index], tag_list[index+1], tag_list[index+2])

# todo: add for word position in lead sentence, word position, first word occurrence position and tfidf and stop
words

return dict_feature

```

tfidf_training

```
# -*- coding: utf-8 -*-

import nltk

import os

import pickle

import sys

import re

from sklearn.feature_extraction.text import TfidfVectorizer

TFIDF_LOCATION = '../model/tfidf.pickle' #this file is genetrated here

# TODO need fixing

def process_input_directory(input_directory):

    all_text = []

    for file_name in os.listdir(input_directory):

        file_path = os.path.join(input_directory, file_name)

        try:

            with open(file_path, 'r') as file:

                lines = file.read()

                all_text.append(lines)

        except:

            print("Error in Read file")

    return all_text

def generate_tf_idf_values(all_text):

    """

    Generate the tfidf values and store it in file.

    """

    global TFIDF_LOCATION

    tfidf = TfidfVectorizer(stop_words='english', smooth_idf=True, encoding='utf-8')

    response = tfidf.fit_transform(all_text)

    feature_names = tfidf.get_feature_names()

    with open(TFIDF_LOCATION, 'w') as out_file:

        for col in response.nonzero()[1]:

            out_file.write('%s %s\n' % (feature_names[col], response[0, col]))
```

```

if __name__ == '__main__':
    path= 'C:/Users/Pankaj Kumar/Desktop/Project/major/major_project/data/segmented/'
    all_text = process_input_directory(path)
    print('Got all data')
    generate_tf_idf_values(all_text)

Title_synthesis_train

import codecs
import os
import pickle
import sys
import nltk
from nltk.classify import maxent
from title.main.content_selection_classify import *
from title.main.featureFunctions.title_model_features import get_title_synthesis_features
from title.main.featureFunctions.BLEU_comparison import get_bleu_score
from title.main.featureFunctions.generate_language_model_features import get_feature_values
title_feature_set = []
title_classifier = None
def get_bleu_score_probability(file_location):
    """For the passed input file, returns the bleu score of the title with reference to the text.
    """
    with open(file_location, 'r') as file:
        all_lines = file.readlines()
        actual_title = all_lines[0]
        all_lines = all_lines[1:]
        bleu_score = get_bleu_score(actual_title, all_lines)
        return bleu_score
def process_directory(input_directory):
    """Processes the entire directory passed as input to generate the feature values.
    """
    global title_feature_set

```

```

all_titles = []

dict_content_score = { }

dict_bleu = { }

error_file = open('Error/title_error.txt', 'w')

for file_name in os.listdir(input_directory):

    try:

        file_path = os.path.join(input_directory, file_name)

        title, word_dict = classify_dev_file(file_path)

        content_score = 0

        for word in title.split():

            # todo: recheck this, what if word is present in title but not in text?

            content_score += word_dict.get(word, 0)

        if content_score in dict_content_score:

            dict_content_score[content_score] += 1

        else:

            dict_content_score[content_score] = 1

        # get bleu score

        bleu_score = get_bleu_score_probability(file_path)

        if bleu_score in dict_bleu:

            dict_bleu[bleu_score] += 1

        else:

            dict_bleu[bleu_score] = 1

        all_titles.append(title)

    except:

        error_file.write('filename : %s\n' % file_name)

        import traceback

        error_file.write(traceback.format_exc())

        error_file.write('\n')

        continue

error_file.close()

```

```

title_feature_set = get_feature_values(all_titles)

for (score, count) in dict_content_score.items():
    output_dict = {'content_score': score}
    outcome = float(count) / len(dict_content_score)
    title_feature_set.append((output_dict, outcome))

for (score, count) in dict_bleu.items():
    output_dict = {'bleu_score': score}
    outcome = float(count) / len(dict_content_score)
    title_feature_set.append((output_dict, outcome))

import json

file = open('temp/sequence.txt', 'w') # this file is missing --resolved

for entry in title_feature_set:
    file.write(json.dumps(entry))
    file.write('\n')

file.close()

def train():
    """Trains the model using the feature set generated above.
    """
    global title_classifier, title_feature_set
    title_classifier = maxent.MaxentClassifier.train(title_feature_set, max_iter=10)

def save():
    """Saves the model so that it can be used without re-running the training part again.
    """
    global title_classifier
    out_file = open('model/title_synthesis.pickle', 'wb')
    pickle.dump(title_classifier, out_file)
    out_file.close()

if __name__ == '__main__':
    initialise() # content_selection_classify.

    path = 'C:/Users/Pankaj Kumar/Desktop/Project/major/major_project/data/out_pos_tagged/'
    process_directory(path)

    train()

```



```
save()
```

Content_selection_classify

```
import math

import operator

import pickle

from title.main.Utils import get_start_end_indices

from title.main.featureFunctions.features import get_feature_dict

from nltk.corpus import stopwords

from title.main.featureFunctions.file_level_features import get_word_range

classifier = None

tfidf_dict = { }

stop_word_list = []

TFIDF_LOCATION = 'model/tfidf.pickle' # this file is missing --resolved

import os

def initialise():

    """Initialises the globally declared variables.

    These variables are used throughout the file.

    """

    global classifier, tfidf_dict, stop_word_list

    with open('model/content_selection.pickle','rb') as file: # this file is missing

        classifier = pickle.load(file)

    with open(TFIDF_LOCATION, 'r') as file:

        for line in file:

            parts = line.strip().split(' ')

            if len(parts) >= 2:

                tfidf_dict[parts[0]] = parts[1]

    stop_word_list = stopwords.words('english')

    return

def get_tfidf_score(all_lines):

    """For an entire file text, returns a dictionary mapping word to range of tf-idf values it belongs to.
```

This information is used as a part of feature function.

```
"""

global tfidf_dict
word_dict = {}

for line in all_lines:

    word_list = line.strip().split()

    for word in word_list:

        word = word.rsplit('/', 1)[0]

        if word in tfidf_dict:

            word_dict[word] = tfidf_dict.get(word)

all_values = list(word_dict.values())
all_values.sort()

length = len(all_values) - 1

first_range_boundary = all_values[int(math.floor(0.9 * length))]
second_range_boundary = all_values[int(math.floor(0.1 * length))]

for (key, value) in word_dict.items():

    if value >= first_range_boundary:

        word_dict[key] = '1_10'

    elif value >= second_range_boundary:

        word_dict[key] = '10_90'

    else:

        word_dict[key] = '90_100'

return word_dict

def get_file_level_details(file_path, headers_present=True):

    """Returns file level feature functions details.
```

These are used as a part of the feature functions for querying the models.

```
"""

global file_level_dict

with open(file_path, 'r') as file:

    lines = file.readlines()
```

```

    all_lines = lines[1:]

    file_level_dict = get_word_range(all_lines)

    word_dict = get_tfidf_score(all_lines)

    return file_level_dict, word_dict

def process_sentence(sentence, file_level_dict, word_dict):
    """For the sentence passed, returns the words along with the probabilities of each word to be present in title.

    Uses the content generation model trained before to get the probability.

    """
    global classifier

    if len(sentence)<=0:

        return

    words = sentence.strip().split()

    title_words = { }

    for index in range(0, len(words)):

        start_index, end_index = get_start_end_indices(index, len(words))

        feature_dict = get_feature_dict(words[start_index: end_index], index-start_index)

        word = words[index].rsplit('/', 1)[0]

        feature_dict['tfidf'] = word_dict.get(word, '90_100')

        feature_dict['lead_sentence'] = file_level_dict[word]['lead_sentence']

        feature_dict['first_occurance'] = file_level_dict[word]['first_occurance']

        feature_dict['range'] = ','.join(str(x) for x in file_level_dict[word]['range'])

        feature_dict['stop_word'] = 1 if word in stop_word_list else 0

        output = classifier.prob_classify(feature_dict)

        if output.prob(1) > 0.0:

            title_words[words[index]] = max(output.prob(1), title_words.get(words[index], 0))

    return title_words

def classify_dev_file(file_location):
    """For given file path, returns a dictionary of all the words along with the probability value.

    This function is called by title synthesis process during training and so return value consists of all the words.

    """

```

```

global classifier

file_level_dict, word_dict = get_file_level_details(file_location)

with open(file_location, 'r') as file:

    sentences = file.readlines()

    actual_title = sentences[0]

    all_potential_title_words = { }

    sentences = sentences[1:]

    for sentence in sentences:

        title_words = process_sentence(sentence, file_level_dict, word_dict)

        if title_words:

            for (key, value) in title_words.items():

                all_potential_title_words[key] = max(value, all_potential_title_words.get(key, 0))

    return actual_title, all_potential_title_words

def classify_new_file(file_path):
    """
    This function is called by decoding algorithm.
    For given input file, it returns a dictionary of 20 words along with their associated probability which are most
    suitable to be included in the title.
    """
    file_level_dict, word_dict = get_file_level_details(file_path, False)

    all_potential_title_words = { }

    with open(file_path, 'r') as file:

        sentences = file.readlines()

        for sentence in sentences:

            title_words = process_sentence(sentence, file_level_dict, word_dict)

            if title_words:

                for (key, value) in title_words.items():

                    all_potential_title_words[key] = max(value, all_potential_title_words.get(key, 0))

    dict_unique_words = { }

```

```

sorted_title_words = sorted(all_potential_title_words.items(), key=operator.itemgetter(1), reverse=True)

top_20_words= {}

for entry in sorted_title_words:

    word_with_tag, value = entry

    word = word_with_tag.rsplit('/', 1)[0]

    if word not in dict_unique_words:

        dict_unique_words[word] = 1

    top_20_words[word_with_tag] = value

    if len(dict_unique_words) > 20:

        break

return top_20_words

```

Content_Selection_train

```

import math

import os

import pickle

import sys

from title.main.featureFunctions.features import get_feature_dict, get_outcome

from title.main.featureFunctions.file_level_features import get_word_range

from nltk.classify import maxent

from nltk.corpus import stopwords

from title.main.Utills import get_start_end_indices

feature_set = []

classifier = None

tfidf_dict = {}

stop_word_list = stopwords.words('english')

TFIDF_LOCATION = 'model/tfidf.pickle' # this file is missing -- resolved

#nltk.config_megam()

def initialise():

    """Initialises the globally declared variables.

    These variables are used throughout the file.

    """

```

```

global tfidf_dict

file = open(TFIDF_LOCATION, 'r')

for line in file:

    parts = line.strip().split(' ')

    if len(parts)>=2:

        tfidf_dict[parts[0]] = parts[1]

return

def get_tfidf_score(all_lines):

    """

    For an entire file text, returns a dictionary mapping word to range of tf-idf values it belongs to.

    This information is used as a part of feature function.

    """

    global tfidf_dict

    word_dict = {}

    for line in all_lines:

        word_list = line.strip().split()

        for word in word_list:

            word = word.rsplit('/', 1)[0]

            if word in tfidf_dict:

                word_dict[word] = tfidf_dict.get(word)

    all_values = list(word_dict.values())

    all_values.sort()

    length = len(all_values) - 1

    first_range_boundary = all_values[int(math.floor(0.9*length))]

    second_range_boundary = all_values[int(math.floor(0.1*length))]

    for (key, value) in word_dict.items():

        if value >= first_range_boundary:

            word_dict[key] = '1_10'

        elif value >= second_range_boundary:

            word_dict[key] = '10_90'

        else:

            word_dict[key] = '90_100'

```

```

    return word_dict

def get_file_level_details(file_path):
    """
    Returns file level feature functions details.
    These are used as a part of the feature functions for querying the models.
    """
    with open(file_path, 'r', encoding='utf-8') as file:
        lines = file.readlines()
        all_lines = lines[1:]
        file_level_dict = get_word_range(all_lines)
        word_dict = get_tfidf_score(all_lines)
    return file_level_dict, word_dict

def process_sentence(sentence, title, file_level_dict, word_dict):
    """
    For the sentence passed, generates the feature sets for all the words present in the sentence.
    The generated feature set is used to train the model.
    """
    global feature_set, stop_word_list
    if len(sentence)<=0:
        return
    words = sentence.strip().split()
    for index in range(0, len(words)):
        start_index, end_index = get_start_end_indices(index, len(words))
        outcome = get_outcome(words[index], title)
        feature_dict = get_feature_dict(words[start_index: end_index], index-start_index)

        # additional fields
        word = words[index].rsplit('/', 1)[0]
        feature_dict['tfidf'] = word_dict.get(word, '90_100')
        feature_dict['lead_sentence'] = file_level_dict[word]['lead_sentence']
        feature_dict['first_occurance'] = file_level_dict[word]['first_occurance']
        feature_dict['range'] = ', '.join(str(x) for x in file_level_dict[word]['range'])

```

```

        feature_dict['stop_word'] = 1 if word in stop_word_list else 0

        # add this to set of all features

        feature_set.append((feature_dict, outcome))

    return

def process_input_directory(directory):
    """
    Processes the entire directory passed as input to generate the feature values.
    """
    count = 0
    with open('Error/error_content_selection.txt', 'w') as error:
        for file_name in os.listdir(directory):
            count += 1
            try:
                file_path = os.path.join(directory, file_name)
                file_level_dict, word_dict = get_file_level_details(file_path)

                with open(file_path, 'r') as file:
                    sentences = file.readlines()
                    title = sentences[0]
                    sentences = sentences[1:]
                    for sentence in sentences:
                        process_sentence(sentence, title, file_level_dict, word_dict)
            except:
                error.write('filename : %s\n' % file_name)
                import traceback
                error.write(traceback.format_exc())
                error.write('\n')
                continue
    return

def train_model():
    """Trains the model using the feature set generated above.
    """

```



```

global classifier, feature_set

#print(feature_set)

classifier = maxent.MaxentClassifier.train(feature_set,max_iter=10)

#classifier = MaxentClassifier.train(feature_set, "megam")

def save_classifier():

    """Saves the model so that it can be used without re-running the training part again.

    """

    global classifier

    out_file = open('model/content_selection.pickle', 'wb') # this file should be created

    pickle.dump(classifier, out_file)

    out_file.close()

if __name__ == '__main__':

    initialise()

    path= 'C:/Users/Pankaj Kumar/Desktop/Project/major/major_project/data/out_pos_tagged/'

    process_input_directory(path)

    train_model()

    save_classifier()

```

Decoding

```

import copy

from heapq import heappush, heappop

from title.main.Utils import remove_tags_from_line

from title.main.content_selection_classify import *

from title.main.title_synthesis_classify import *

logger = None

LOG_FILE_LOCATION = 'temp/parse_log.log' # this file is missing --resolved

def initialise_all():

    """

    Initialise the content selection and title synthesis models

    """

    global logger

    initialise()

    title_synthesis_initialise()

```

```

logger = open(LOG_FILE_LOCATION, 'w', encoding='utf-8')

def get_file_headings(file_path, title_length=8):
    """
    Creates the actual headings by parsing the passed file and generating sequences.
    """

    global logger

    top_sentence_list = []

    with open(file_path, 'r') as file:
        text = file.read()

    top_20_words = classify_new_file(file_path)

    heap, next_heap = [], []

    for word in top_20_words:
        heappush(heap, (0, [word]))

    index = 0
    max_length = 20

    while index < max_length:
        if index < 1:
            max_range = 21
        elif index < 3:
            max_range = 3
        else:
            max_range = 2

        index2 = 1
        probability_list = []

        while heap and index2 < max_range:
            prob, word = heappop(heap)

            if prob not in probability_list:
                probability_list.append(prob)

            index2 += 1

        for all_word in top_20_words:
            if all_word not in word:
                word_copy = copy.deepcopy(word)

```

```

existing_words = [word1.rsplit('/', 1)[0] for word1 in word_copy]

if all_word.rsplit('/', 1)[0] in existing_words:
    continue

word_copy.append(all_word)

word_str = ' '.join(word_copy)

probab_value = get_title_synthesis_score(word_str, top_20_words, text)

logger.write('%s- %s\n' %(word_str, probab_value))

heappush(next_heap, (-1*probab_value, word_copy))

heap = next_heap

next_heap = heap

index += 1

max_length = title_length # change title length if needed

count = 0

while heap and count < 10:

    count += 1

    probab, sentence = heappop(heap)

    top_sentence_list.append(remove_tags_from_line(sentence))

return top_sentence_list

```

Dev_test

```

import os

from title.main.decoding import initialise_all, get_file_headings

from title.main.Utils import remove_tags_from_line

from title.POSTagger import pos_tagger

def get_file_path(input_path):
    """
    Creates a temp file which maintains the format in which decoding algorithm implements it.
    """

    temp_location = 'temp/trial.txt'

    with open(temp_location, 'w') as file:

        out_file = file

    with open(input_path, 'r') as in_file:

        lines = in_file.readlines()

```

```

        title = lines[0] # actual title

        data = lines[1:]

        out_file.writelines(data)

    title = remove_tags_from_line(title.split())

    return title, temp_location

def process_directory(input_dir, output_dir):
    """
    Processes all the files in input directory and writes the output to a different directory
    """
    for file_name in os.listdir(input_dir):
        output_file = os.path.join(output_dir, 's_processed.txt' % file_name.split('.')[0])
        with open(output_file, 'w', encoding='utf-8') as out_file:
            title, file_path = get_file_path(os.path.join(input_dir, file_name))
            top_sentences = get_file_headings(file_path, len(title.split()))
            out_file.write('%s\n' % title.strip())
            sentences = '\n'.join(top_sentences)
            out_file.write(sentences)

if __name__ == '__main__':
    os.chdir('../../title/main/')
    initialise_all()
    #os.chdir("../../RDRPOSTagger")
    in_path = 'C:/Users/Pankaj Kumar/Desktop/Project/major/major_project/data/result/test_input/'
    seg_out_path = 'C:/Users/Pankaj Kumar/Desktop/Project/major/major_project/data/result/segmented/'
    out_path = 'C:/Users/Pankaj Kumar/Desktop/Project/major/major_project/data/result/test_output/'
    #pos_tagger.parse_directory(in_path,seg_out_path)
    #os.chdir('../../title/main/')
    process_directory(seg_out_path,out_path)

```

Title_synthesis_classify

```

"""This files contains functions for getting the result from title synthesis model.
"""

import pickle

```

```

from title.main.featureFunctions.BLEU_comparison import get_bleu_score

from title.main.featureFunctions.title_model_features import get_classification_dictionary

from title.main.featureFunctions.generate_language_model_features import get_features

title_synthesis_classifier = None

def title_synthesis_initialise():

    """Loads the classifier object with the contents of stored model file.

    """

    global title_synthesis_classifier

    file = open('model/title_synthesis.pickle','rb')  #this file is missing --resolved

    title_synthesis_classifier = pickle.load(file)

    file.close()

def get_title_synthesis_score(title_seq, dict_content_score, file_text):

    """For a title sequence given the entire file text, returns the probability of that sequence to be the title.

    Uses the trained title synthesis model to find out the probability.

    """

    global title_synthesis_classifier

    title_words = title_seq.split()

    feature_list = get_features(title_seq)

    # handle the case of start tag for trigram

    start_feature = feature_list.pop(0)

    start_feature['content_score'] = dict_content_score[title_words[0]] + dict_content_score[title_words[1]]

    start_feature['bleu_score'] = 0.0

    probability = title_synthesis_classifier.classify(start_feature)

    if probability == 1.0:

        probability = 0

    for index in range(0, len(title_words)-2):

        title_seq = ''.join(title_words[index: index+3])

        bleu_score = get_bleu_score(title_seq, file_text)

        content_score = 0

        for word in title_words[index: index+3]:

```

```

        content_score += dict_content_score[word]

feature_dict = feature_list.pop(0)
feature_dict['content_score'] = content_score
feature_dict['bleu_score'] = bleu_score
temp = title_synthesis_classifier.classify(feature_dict)
if temp == 1.0:
    temp = 0
probability += temp
return probability

```

Generate_Language_Model_Features

```

import nltk

def increment_key(dict_obj, key):
    if key in dict_obj:
        dict_obj[key] += 1
    else:
        dict_obj[key] = 1

def get_feature_values(all_sentences):
    all_tag_sequence = []
    all_word_sequence = []
    all_feature_list = []
    dict_word = { }

    for sentence in all_sentences:
        sentence = 'start/start %s' % sentence
        words = sentence.split() #word-pos
        tag_list = []
        word_list = []

        for entry in words:
            word, tag = entry.rsplit('/', 1)
            tag_list.append(tag)

            if word not in ['start']:
                word_list.append(word)

```

```

        increment_key(dict_word, word)

    all_tag_sequence.append(tag_list)
    all_word_sequence.append(word_list)

# all_tag_sequence now has a list of tags corresponding to different sentences

bigram_pos_list = []
trigram_pos_list = []
for tag_seq in all_tag_sequence:
    bigram_pos_list.extend(list(nltk.bigrams(tag_seq)))
    trigram_pos_list.extend(list(nltk.trigrams(tag_seq)))

dict_pos_bigram = {}
dict_pos_trigram = {}

for entry in bigram_pos_list:
    increment_key(dict_pos_bigram, entry)

for entry in trigram_pos_list:
    increment_key(dict_pos_trigram, entry)

for (key, value) in dict_pos_trigram.items():
    prob_val = float(value)/dict_pos_bigram[(key[0], key[1])]
    key_str = '_'.join(key)
    dict_temp = {'pos_trigram': key_str}
    all_feature_list.append((dict_temp, prob_val))

# calculate word bigrams now
bigram_word_list = []
for word_seq in all_word_sequence:
    bigram_word_list.extend(list(nltk.bigrams(word_seq)))

```

```

dict_word_bigram = { }

for entry in bigram_word_list:

    increment_key(dict_word_bigram, entry)


# for key, value in dict_word_bigram.iteritems():
#     prob_val = float(value)/dict_word[key[0]]
#     key_str = '_'.join(key)
#     dict_temp = {'word_bigram': key_str}
#     all_feature_list.append((dict_temp, prob_val))


return all_feature_list


def get_features(word_seq):

    word_seq = 'start/start %s'%(word_seq)

    words = word_seq.split()

    all_features = []

    tag_list = []

    word_list = []

    for entry in words:

        word, tag = entry.rsplit('/', 1) #word-pos_tag

        tag_list.append(tag)

        word_list.append(word)


    trigram_list = list(nltk.trigrams(tag_list))

    for entry in trigram_list:

        key_str = '_'.join(entry)

        dict_temp = {'pos_trigram': key_str}

        all_features.append(dict_temp)


    #bigram_list = list(nltk.bigrams(tag_list))

    #for entry in bigram_list:

```



```
# key_str = '_' .join(entry)

# dict_temp = {'word_bigram': key_str}

# all_features.append(dict_temp)
```

```
return all_features
```

Title_Model_Features

```
# define all the feature functions here
```

```
"""
```

```
List of features used:
```

1. Language model features
2. Title Length feature
3. Part of Speech Language Model Feature
4. N-Gram Match feature
5. Content selection feature

```
"""
```

```
import math
```

```
# model feature tuple
```

```
feature_values = []
```

```
# structures for title length feature
```

```
Unique_length_count = 0 # range considered is 3 to 15(13 values )
```

```
title_length_count = { }
```

```
title_length_probability = { }
```

```
# structures for language model feature
```

```
unique_bigram_count = 0
```

```
language_model_count = { }
```

```
language_model_probability = { }
```

```
word_count = { }
```

```
# structure for pos title bigram feature
```

```
unique_bigram_pos_count = 0
```

```
bigram_model_count = { }
```

```
# structure for pos title trigram feature
```

```

unique_trigram_pos_count = 0

trigram_model_count = {}

trigram_model_probability = {}

```

```

def compute_title_length_counts(title_word_tag_list):
    """
    Input: A list with each entry having title from input corpus
    operation: computes the title length for each article title and stores the probability
    for each title length in a length dictionary
    """

    global title_length_count, Unique_length_count # range considered is 3 to 15(13)
    for title in title_word_tag_list:
        count = 0
        tokens = title.split(" ")
        for entry in tokens:
            word, tag = entry.rsplit('/', 1)
            count += 1

        if count in title_length_count:
            title_length_count[count] += 1
        else:
            title_length_count[count] = 1
            Unique_length_count += 1

```

```

def compute_title_length_probability(title_word_tag_list):
    """
    Input: A list with each entry having title from input corpus
    operation: computes the title length for each article title and stores the probability
    for each title length in a length dictionary
    """

    global title_length_count, Unique_length_count, title_length_probability # range considered is 3 to 15(13)
    title_length_probability = title_length_count.copy()

```

```

# computing total title count

total_no_of_titles = len(title_word_tag_list)

compute_title_length_counts(title_word_tag_list)


for i in title_length_count:

    # print "key:"+str(i)+" val:"+str(title_length_count[i])

    temp = (title_length_count[i]) / float(total_no_of_titles)

    # print "temp:"+str(temp)

    title_length_probability[i] = temp

def compute_word_count(title_word_tag_list):

    """

    Input: A list with each entry having title from input corpus
    operation: computes the word count of each word in dataset
    """

    global word_count

    for title in title_word_tag_list:

        tokens = title.split(" ")

        for entry in tokens:

            word, tag = entry.rsplit('/', 1)

            if word in word_count:

                word_count[word] += 1

            else:

                word_count[word] = 1

def compute_language_model_counts(title_word_tag_list):

    """

    Input: A list with each entry having title from input corpus

    operation: computes the language model counts for each article bigram and stores the probability for each
    bigram in dictionary
    """

    global language_model_count, unique_bigram_count

    for title in title_word_tag_list:

        tokens = title.split(" ")

        prev = "start"

```

```

cur = "start"

for entry in tokens:
    word, tag = entry.rsplit('/', 1)

    prev = cur
    cur = word

    if prev in language_model_count:
        if cur in language_model_count[prev]:
            language_model_count[prev][cur] += 1
        else:
            language_model_count[prev][cur] = 1
    else:
        language_model_count[prev] = {}
        if cur in language_model_count[prev]:
            language_model_count[prev][cur] += 1
        else:
            language_model_count[prev][cur] = 1

def compute_language_model_probability(title_word_tag_list):
    """
    Input: A list with each entry having title from input corpus

    operation: computes the language model probability for each article bigram and stores the probability for
    each bigram in dictionary
    """

    global language_model_count, unique_bigram_count, language_model_probability, word_count
    compute_word_count(title_word_tag_list)
    compute_language_model_counts(title_word_tag_list)
    language_model_probability = dict(language_model_count)

    for title in title_word_tag_list:
        tokens = title.split(" ")

        prev = "start"
        cur = "start"

        mycount = 0

        for entry in tokens:

```

```

mycount = mycount + 1

word, tag = entry.rsplit('/', 1)

prev = cur

cur = word

if mycount >= 2:

    if prev in language_model_probability:

        if cur in language_model_probability[prev]:

            language_model_probability[prev][cur] = (language_model_count[prev][cur]) / float(

                word_count[prev])

        else:

            language_model_probability[prev][cur] = (language_model_count[prev][cur]) / float(

                word_count[prev])

    else:

        language_model_probability[prev] = {}

        language_model_probability[prev][cur] = (1) / float(word_count[prev])

def compute_bigram_counts(title_word_tag_list):

    """

    Input: A list with each entry having title from input corpus

    operation: computes the language model counts for each article bigram and stores the probability for each
    bigram POSin dictionary

    """

    global bigram_model_count, unique_bigram_pos_count

    for title in title_word_tag_list:

        tokens = title.split(" ")

        prev = "start"

        cur = "start"

        for entry in tokens:

            word, tag = entry.rsplit('/', 1)

            prev = cur

            cur = tag

            if prev in bigram_model_count:

                if cur in bigram_model_count[prev]:

```

```

        bigram_model_count[prev][cur] += 1
    else:
        bigram_model_count[prev][cur] = 1
    else:
        bigram_model_count[prev] = {}
        if cur in bigram_model_count[prev]:
            bigram_model_count[prev][cur] += 1
        else:
            bigram_model_count[prev][cur] = 1

def compute_trigram_counts(title_word_tag_list):
    """
    Input: A list with each entry having title from input corpus

    operation: computes the language model counts for each article trigram and stores the probability for each
    trigram POS in dictionary
    """

    global trigram_model_count, unique_trigram_pos_count
    local_trigram_count = 0
    lc = 0
    # print title_word_tag_list
    for title in title_word_tag_list:
        local_trigram_count += 1
        # print str(local_trigram_count)+"current line:"+title
        tokens = title.split(" ")
        prev = "start"
        cur = "start"
        next = "start"
        # print "line:"+title
        for entry in tokens:
            word, tag = entry.rsplit('/', 1)

            prev = cur
            cur = next
            next = tag

```

```

# print "LC:"+str(lc)

if prev in trigram_model_count:
    # print "in if"
    if cur in trigram_model_count[prev]:
        # print "in if if"
        if next in trigram_model_count[prev][cur]:
            # print "in if if if"
            trigram_model_count[prev][cur][next] += 1
            # print "dict:"+str(trigram_model_count)
        else:
            # unique_trigram_pos_count = unique_trigram_pos_count+1
            # print "in if if else"
            trigram_model_count[prev][cur] = { }
            trigram_model_count[prev][cur][next] = 1
            # print "dict:"+str(trigram_model_count)

    else:
        # print "in if else"
        trigram_model_count[prev][cur] = { }
        trigram_model_count[prev][cur][next] = 1
        # print "dict:"+str(trigram_model_count)

else:
    # print "in else"
    trigram_model_count[prev] = { }
    trigram_model_count[prev][cur] = { }
    trigram_model_count[prev][cur][next] = 1
    # print "dict:"+str(trigram_model_count)

lc += 1

# print "#####"
```

$P(w_i | w_{i-1} w_{i-2}) = \text{count}(w_i, w_{i-1}, w_{i-2}) / \text{count}(w_{i-1}, w_{i-2})$

```
def compute_pos_language_model(title_word_tag_list):
    """ Input: A list with each entry having title from input corpus
        operation: computes the language model probability for each trigrams of POS and stores the probability for
        each trigram POS in dictionary
    """

    global trigram_model_probability, trigram_model_count, trigram_model_probability, bigram_model_count

    compute_trigram_counts(title_word_tag_list)

    compute_bigram_counts(title_word_tag_list)


    trigram_model_probability = dict(trigram_model_count)

    # computes POS language model probability
    for title in title_word_tag_list:

        prev = "start"

        cur = "start"

        next = "start"

        tokens = title.split(" ")

        mycount = 0

        for entry in tokens:

            word, tag = entry.rsplit('/', 1)

            prev = cur

            cur = next

            next = tag

            # print "&&&&&&&&&&&&&&&&&&&&&&&&&&&&&"

            # print "prev:"+prev+" cur:"+cur+" next:"+next+"my count:"+str(mycount)


        if mycount > 1:

            if prev in trigram_model_probability:

                if cur in trigram_model_probability[prev]:

                    if next in trigram_model_probability[prev][cur]:

                        temp = (trigram_model_count[prev][cur][next]) / float(bigram_model_count[prev][cur])

                        trigram_model_probability[prev][cur][next] = temp

                    mycount += 1
```



```

def compute_POS_language_feature(title_word_tag_list):
    """ Returns POS language model feature value for the title
    """
    global trigram_model_probability
    POSLM_feature = 0
    prev = "start"
    cur = "start"
    next = "start"
    count = 1

    # initialization of dictionary
    tokens = title_word_tag_list.split(" ")
    for entry in tokens:
        word, tag = entry.rsplit('/', 1)
        prev = cur
        cur = next
        next = tag
        count += 1
        # print "prev:"+prev+" cur:"+cur+" next:"+next
        if count > 2:
            if prev in trigram_model_probability:
                if cur in trigram_model_probability[prev]:
                    if next in trigram_model_probability[prev][cur]:
                        probability = trigram_model_probability[prev][cur][next]
                        POSLM_feature = POSLM_feature + math.log(probability, 10)
    return POSLM_feature

def compute_title_length_feature(title_word_tag_list):
    """
    computes the log probability of particular title length and returns the value
    """
    global title_length_probability
    Length_feature = 0

```

```

count = 0

total_no_of_titles = 0

for i in title_length_count:

    total_no_of_titles = total_no_of_titles + title_length_count[i]


tokens = title_word_tag_list.split(" ")

for entry in tokens:

    word, tag = entry.rsplit('/', 1)

    count = count + 1

if count in title_length_probability:

    Length_feature = math.log(title_length_probability[count], 10)

else:

    temp = 1 / float(total_no_of_titles)

    Length_feature = math.log(temp, 10)

return Length_feature

def compute_language_model_feature(title):

    """Returns the language model feature value

    """

    global language_model_probability, word_count

    total_word_count = 0

    for i in word_count:

        total_word_count = total_word_count + word_count[i]

        # print "total word count:" + str(total_word_count)

    prev = "start"

    cur = "start"

    title = title.strip()

    WordOfLine = title.split()

    mycount = 0

    LM_value = 0

    tokens = title.split(" ")

    for entry in tokens:

        word, tag = entry.rsplit('/', 1)

```

```

prev = cur

cur = word

# print "in prev:"+prev+"cur:"+cur+"count:"+str(mycount)

mycount += 1

if mycount > 1:

    if prev in language_model_probability:

        if cur in language_model_probability[prev]:

            LM_value = LM_value + math.log(language_model_probability[prev][cur], 10)

            # print "log LMvalue:"+str(math.log(language_model_probability[prev][cur],10))

            # if word given previous word probability does not exist we use others value as smoothing measure

        else:

            temp = 1 / (word_count[prev])

            LM_value += math.log(temp, 10)

    else:

        temp = 1 / (total_word_count)

        LM_value += math.log(temp, 10)

return LM_value

def get_title_synthesis_features(title_word_tag_list):

    """

    calls the feature functions and stores feature values for the model in a tuple in the format

    ({POSLM:"NN VB NN"},outcome,{title_len,len_val},outcome,

    """

    global title_length_probability, language_model_probability, trigram_model_probability, feature_values

    compute_pos_language_model(title_word_tag_list)

    compute_language_model_probability(title_word_tag_list)

    compute_title_length_probability(title_word_tag_list)

    # adding all features of one title

    for head in title_word_tag_list:

        #

        "##### print

        # print "line:"+head

        count = 0

```

```

tokens = head.split(" ")

# adding feature 1
for entry in tokens:
    word, tag = entry.rsplit('/', 1)
    count += 1
temp_dict = {'title_len': count}

feature1 = (temp_dict, title_length_probability[count])
# print "current f1 value:"+str(feature1)
feature_values.append(feature1)

# adding feature 2
POSLM_feature = compute_POS_language_feature(head)
pos_string = ""
temp_dict = { }
# print "pos tag string:"+pos_string
# initialization of dictionary
# tokens = head.split(" ")
for entry in tokens:
    word, tag = entry.rsplit('/', 1)
    pos_string = pos_string + tag + " "
temp_dict['pos_LM'] = pos_string
feature2 = (temp_dict, POSLM_feature)
feature_values.append(feature2)
# print temp_dict
# print "current f2 value:"+str(feature2)

# adding feature 3
LM_feature = compute_language_model_feature(head)
word_bigram_string = ""
temp_dict = { }
# print "word string:"+word_bigram_string
prev = "start"
cur = "start"

```

```

lm = 0

# tokens = head.split(" ")
for entry in tokens:
    word, tag = entry.rsplit('/', 1)
    prev = cur
    cur = word
    word_bigram_string = word_bigram_string + " " + prev + "-" + cur

# print "word string:"+word_bigram_string
LMvalue = compute_language_model_feature(head)
temp_dict['LM'] = word_bigram_string
feature3 = (temp_dict, LMvalue)
feature_values.append(feature3)
# print "current f3 value:"+str(feature3)

del feature1
del feature2
del feature3

# print feature_values
return feature_values

def get_classification_dictionary(title_word_tag_list):
    """
    returns the list of the form
    {'title_len': 3}, ({'pos_LM': 'NN VV MM '},{ 'LM': ' start-a a-b b-c'},
    """
    local_dict = {}
    count = 0
    tokens = title_word_tag_list.split(" ")
    # adding feature 1
    for entry in tokens:
        count += 1
    local_dict['title_len'] = count

```

```

# adding feature 2
pos_string = ""
for entry in tokens:
    word, tag = entry.rsplit('/', 1)
    pos_string = pos_string + tag + " "
local_dict['pos_LM'] = pos_string
word_bigram_string = ""
prev = "start"
cur = "start"
lm = 0
for entry in tokens:
    word, tag = entry.rsplit('/', 1)
    prev = cur
    cur = word
    word_bigram_string = word_bigram_string + " " + prev + "-" + cur
local_dict['LM'] = word_bigram_string
return local_dict

```

BLEU

```

# -*- coding: utf-8 -*-

from nltk.translate.bleu_score import sentence_bleu as bleu

# Human evaluations of machine translation outputs require considerable effort and are expensive.
# Human evaluations can take days or even weeks to finish so a new scoring system was developed to
# automate this process of evaluation. This method is commonly referred to as BLEU Score

def get_bleu_score(candidate_text, full_text, N=3):
    all_words = []
    for line in full_text:
        words = line.split() # word/pos-tag pair
        for word in words:
            word = word.rsplit('/', 1)[0]
            all_words.append(word)

```

```

weight = 1.0 / N
bleu_score = 0.0
candidate_seq = candidate_text.split()
candidate_seq = [word.rsplit('/', 1)[0] for word in candidate_seq]
for index in range(len(candidate_seq) - 2):
    bleu_score += bleu([all_words], candidate_seq[index: index + 3], [weight])
return bleu_score

```

Evaluation:

```

def evaluation_cosine(s1, s2):
    """
    Returns cosine and tf idf evaluation for two given sentences
    """
    words = {}
    i = 0
    # loop through each list, find distinct words and map them to a
    # unique number starting at zero

    for word in s1:
        if word not in words:
            words[word] = i
            i += 1

    for word in s2:
        if word not in words:
            words[word] = i
            i += 1

    # create a numpy array (vector) for each input list, filled with zeros
    word_list = list(words.keys())
    a = np.zeros(len(word_list))
    b = np.zeros(len(word_list))

```

```

# loop through each list and create a corresponding vector for it
# this vector counts occurrences of each word in the dictionary
for word in s1:
    index = words[word]
    a[index] += 1

for word in s2:
    index = words[word]
    b[index] += 1

# use numpy's dot product to calculate the cosine similarity
sim = np.dot(a, b) / np.sqrt(np.dot(a, a) * np.dot(b, b))
# print "cosine value is "+str(sim)
return sim # similarity score of to title

def evaluate_title(actual,generated):
    cosine = 0.0
    max_cosine = 0.0
    max_cosine_file = 0
    max_cosine = 0
    for i in range(len(generated)):
        current = generated[i]
        cosine = evaluation_cosine(actual, current)
        if (max_cosine < cosine):
            max_cosine = cosine
            max_cosine_file = i
    return max_cosine_file,max_cosine

```


6. CONCLUSION

Web Document clustering is an exciting and thriving research area. In particular, unsupervised clustering of web documents so far less studied than supervised learning in this context has many future applications in organising and understanding the WWW, as well as other corpora of text. We have done data processing and document classification which is giving average 96 percent accuracy.

We have presented a system that automatically generates a title for a single document. This uses a mixed approach of *sentence extraction* and *machine learning*. This hybrid approach enabled us to produce title shorter than a single sentence. Also, this double process produces a good balance between informativeness, compression and readability of the generated title.

In this work we presented a maximum entropy discriminative framework for the title generation task with three principal components : 1) A Content Selection Model that uses a rich feature set comprising word and POS n-gram features, positional features and word frequency features. 2) A Title Synthesis Model that combines multiple knowledge sources (models) as feature functions into a second Maximum Entropy model and is used to score candidate title sequences and 3) A decoding algorithm that uses beam search pruning to explore the title hypothesis space and generate the optimal title sentences . Within our framework, choice of title words is not restricted to only the words present within the story. This is made possible through the use of a word translation model that given sufficient training data learns logical word substitutions for story words.

REFERENCES:

- [1] "Unsupervised Clustering" Mark P. Sinka, David W. Corne Department of Computer Science, University of Reading, Reading, RG6 6AY, UK m.p.sinka@reading.ac.uk, d.w.corne@reading.ac.uk;pp
- [2] "Probabilistic Latent Semantic" Thomas Hofmann EECS Department, Computer Science Division, University of California, Berkeley & International Computer Science Institute, Berkeley, CA hofmann@cs.berkeley.edu
- [3] "Text condensation as knowledge base abstraction" U. Reimer and U. Hahn, In *the Proceedings of Fourth Conference on Artificial Intelligence Applications*, pages 338–344, 1988.
- [4] "Title generation based on statistical translation" M. Banko, V. Mittal, and M. Witbrock, In *the Proceedings of Association for Computational Linguistics*, 2000.
- [5] "Hedge trimmer: a parse and trim approach to title generation" Bonnie Dorr, David Zajic, and Richard Schwartz, In *Proceedings of the HLT-NAACL 03 on Text summarization workshop*, pages 1–8, Morristown, NJ, USA, 2003, Association for Computational Linguistics.
- [6] "Improved Algorithms For Keyword Extraction And Title Generation From Unstructured Text." Mondal, Amit Kumar, and Dipak Kumar Maji, *CLEAR Journal* (2013)