UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Latent semantic models: A study of probabilistic models for text in information retrieval

By

Siyabonga Zimozoxolo Mjali

Submitted in partial fulfilment of the requirements for the degree

Magister Scientiae in Mathematical Statistics

in the

In the Department of Statistics

In the Faculty of

Natural and Agricultural Sciences

at the

Univeristy of Pretoria

March 31, 2020

I, *Siyabonga Zimozoxolo Mjali*, declare that this dissertation, which I hereby submit for the degree Magister Scientiae in Mathematical Statistics at the University of Pretoria, is my own work and has not previously been submitted by me for a degree at this or any other tertiary institution.

Signature:

Date:

ABSTRACT

Large volumes of text is being generated every minute which necessitates effective and robust tools to retrieve relevant information. Supervised learning approaches have been explored extensively for this task, but it is difficult to secure large collections of labelled data to train this set of models. Since a supervised approach is too expensive in terms of annotating data, we consider unsupervised methods such as topic models and word embeddings in order to represent corpora in lower dimensional semantic spaces. Furthermore, we investigate different distance measures to capture similarity between indexed documents based on their semantic distributions. These include cosine, soft cosine and Jensen-Shannon similarities. This collection of methods discussed in this work allows for the unsupervised association of semantic similar texts which has a wide range of applications such as fake news detection, sociolinguistics and sentiment analysis.

Acknowledgement

The financial assistance of the Center of Artificial Intelligence Research (CAIR) and The HUB Internship towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the authors and are not necessarily to be attributed to CAIR or The HUB internship.

Contents

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In the age of big data, many businesses suffer from collecting large repositories of data and not being able to structure them in order to retrieve useful information or generate insight. A common way of representing text data in a vector format is the Vector Space Model (VSM) [53]. In the Vector Space Model documents are represented as vectors in a high dimensional vector space [45]. The high dimensional space is a result of the various ways features are generated for text data namely; words, n-grams, parts-of-speech etc. In many cases humans can easily understand the intended meaning of a word, however this is not so simple to do computationally [22]. Take for example words that are similar for a given context but different enough to be considered different features, such as, 'play' and 'game'. They will be indexed as two separate features, each one represented by its own dimension in the vector space, yet they are semantically related [53] . In [13], Deerwester (1990) introduced an approach to automatically index documents, using words as features. The method is aimed at addressing polysemy and synonymy in the task of Information Retrieval (IR) by using Singular Vector Decomposition (SVD). The high dimensional structure, the original document-word matrix, is reduced into a lower dimensional matrix that represents semantic relatedness of each word to some underlying variable. The problem with this approach is that the query may have 1) features not considered in the index, or 2) the user may be using synonyms to index terms. In both cases if a document is conceptually related to the query this will not be reflected in the measures of similarity. The most common application of VSM is the bag-of-words approach [61]. To model semantic relationships, researchers have resorted to topic models [9, 13] and word embeddings [33, 34] to enhance document representation [61]. We investigate LDA and word2vec models on a relevance judgement tasks by observing the performance through similarity measures namely: cosine similarity, soft cosine similarity and the Jensen-Shannon divergence measure.

1.1 Objective

We explore the LDA and word2vec on the task of retrieving relevant document using semantic similarity. We use soft cosine and Jensen-Shannon divergence for comparing a query to the corpus. The objective is to determine if the LDA captures semantic representation comparatively better than the word2vec on relatively small corpora and this difference can be quantified in semantic representation terms.

1.2 Motivation

For retrieval purposes we will observe the similarity measures for each of the different models as well as other evaluation measures for the purpose of identifying a model that will perform well in this task of relevance judgement when we have taken into account semantic similarity. This will allow for a better retrieval performance which has many applications in IR and document classification.

1.3 Dissertation structure

The dissertation structure is as follow:

- Chapter 2 is a review of literature in information retrieval and the connection of probabilistic topic models to this task.
- Chapter 3 discusses Vector Space Model frameworks defined to apply mixture models on text data, transformation measures and similarity metrics.
- Chapter 4 Bayesian learning for text analytics where concept learning for machines is discussed and the derivations for the naive Bayes are made.
- Chapter 5 Latent Variable Models give a discussion the two models implemented for this study namely; Latent Dirichlet Allocation and probabilistic Latent Semantic Indexing.
- Chapter 6 Application reports on results of the two models and discusses interpretation.

Chapter 2

Literature Review

Today we have been blessed and cursed with an overload of information [2]. The vast amount of text data generated everyday has been growing at alarming rates. With increasing amounts of data generation over shorts spans of time, machine learning methods are required to process and analyze these large volumes of data for insight generation. Across all multimedia, text data occurs in the largest volumes[1] through the publication of newspapers, blog posts, emails, phone texts, social media posts (Tweets, Facebook posts and Instagram captions), forums, question-answer sites and research articles. These texts are unstructured, making it difficult to easily process and visualise. The reasons stated partially motivate the work in this report and contribute in the area of text analysis. Text analytics, or text mining is the process of deriving insight and discovering hidden structures from text data. This includes tasks such as text classification [39], information retrieval [49, 50] and topic models [9, 12, 30]. In machine learning , there are three types of machine learning approaches:

1. supervised,
2. semi-supervised, and
3. unsupervised.

For supervised tasks, the goal is to learn a mapping from inputs x to outputs y. In layman's terms, the practitioner deals with labelled data, meaning each observation in the training set has a known label (or output). Tasks such as text classification fall in this sphere of machine learning. The challenge, however, with this approach in the context of semantic similarity is the cost of labelled data. For supervised learning, data are usually labelled manually, which implies expensive resources.

Semi-supervised tasks involve data sets which contain small sets of labelled data and some discovery must be made subsequently. Though this set of approaches

[1] https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#6e8cce2e60ba

made improvements in semantic role labelling [16], we do not consider them for this study.

Finally, unsupervised learning can be described as a descriptive or exploratory method. Here we have some input x and the goal is to elucidate meaningful patterns [37]. This is also known as knowledge discovery. The problems in the unsupervised space are less well-defined, since no ground truth exists. Nonetheless, these approaches provides us with an array of useful techniques since in text mining applications, little or no prior knowledge is available about the content of text data. This calls for unsupervised methods to structure and relate text sources to each other [12]. We consider this set of methods for the task of semantic representation of large collections of text, as it is able to retrieve relevant documents based on a query's semantic similarity to the collection. This approach was considered in [61], where they identified that document representation is important for retrieval and proposed the LDA to capture important relationships between words. However, the consideration made in [61] are in the language modelling context, where we will observe the statistical properties of semantic relationships between a query and a reference corpus.

This chapter provides an overview of unsupervised methods relevant to latent semantic representation and analysis of text data.

2.1 Algebraic Approaches

Latent Semantic Indexing (LSI) [13] and Latent Semantic Analysis (LSA) [25] were developed to tackle two inherent issues in the information retrieval process, namely synonymy and polysemy by using singular value decomposition (SVD). SVD produces a lower dimensional vector space which can be used to determine semantic properties between documents and words [18]. This method performs better than term-matching methods [13, 19] like full text scanning which relies on the sub-string test, a method that goes through all documents to find the specified string to determine relevance of documents in the retrieval process [15]. The SVD method produces a correlations matrix between words and documents [31]. As a result semantic relationships are better represented by the LSI than the lexical matching approaches. These algebraic methods also address issues such as dimensionality. As a consequence we are able to process large quantities of data, specifically for text that requires many features to train models. The LSI is described as a feature extraction method in [44], where linear combinations of the original features are used instead of the original features. As a result the number of extracted features will generally be significantly less than the original feature set. Through these transformations dimensionality and sparsity are addressed. In this study we consider a probabilistic approach to the study of semantic similarity and relevance judgement. We discuss these methods next.

2.2 Probabilistic Approaches

Though the LSI and LSA methods have contributed greatly in the introduction of reduced dimensional spaces for indexing, the task of IR requires more probabilistic approaches to semantic representation than algebraic methods. The probabilistic approach to semantic representation is reflected in better retrieval performance over standard raw term frequency approaches and LSI [19]. The probabilistic foundation in these methods is established by mixture models. Topic models make use of mixture models and probabilistic generative assumptions to introduce underlying thematic structures for each document in a corpus using latent variables [18]. The distinct advantage of topic models over other semantic representation methods is the formation of word clusters, since each topic is a distribution of words, which are often correlated and reveal underlying themes. Topic models are applicable to many areas of natural language processing such as information retrieval [61], collaborative filtering [59], document classification [48], word sense disambiguation [42], and domain modelling [11].

Topic models are unsupervised, meaning no unique parameterisation exists to explain the ground truth for some given phenomenon [37]. The probabilistic latent semantic indexing model (pLSI) [19] is based on the likelihood principle and defines a generative model for the data using a mixture of Multinomials to describe word samples for each document. Each of the mixture components can be thought of as a topic. The model has been criticized for not making assumptions about how the mixture weights of a topic are generated for each document. This lack of generalisations for unseen documents leads to over-fitting and bad performance on out of sample test data when looking at perplexity. The LDA (Latent Dirichlet Allocation) [18] corrects this over-fitting problem by attaching a Dirichlet prior, a conjugate prior to the multinomial, to the topic distributions for some arbitrary document.

The LDA [9] is a generative model that has seen success in many information retrieval tasks [60, 61]. In [9] the goal of topic modelling is document generalisation. Document generalisation can be achieved by finding the underlying semantic context which is represented by the words of a document. As a result topic models illustrate how using a different representation can provide new insight into statistical modelling of language [18]. We plan to exploit those insights to represent corpora and use underlying semantic relationships between words and documents for performance review in our information retrieval exercise.

2.3 Word Embeddings

The two methods above consider the bag-of-words representation of a corpus. This implies count statistics as the modelling premises as the frequency of a word represents its occurrence in a document. Count-based methods offer a lot in terms of simplicity and robustness but discard word order. An alternative to count-based semantic representations is prediction-based representations. In these methods, weights in a word vector directly maximize the probability of the contexts [4, 34]. In [34] a distributed representation approach [20] is considered by making of a single layer neural network to represent words as high quality vectors [5]. The basic idea is to train a single layer neural network to be able to predict a word by the words around it. The word2vec model is an example of this and has shown to be a state of the art in various NLP tasks. Since this model captures similarity between words beyond syntactic rules [33] we use this model to uncover semantic similarity at a word level and compare the retrieval results to those of the topic models.

2.4 Evaluation

The output of any latent semantic representation is a set of vectors - whether topic distributions or dense word embeddings. It is important to know if these latent representations describe the entire corpus. For different approaches, different evaluation methods exist.

2.4.1 Topic Model Evaluation

We must evaluate how well the topic models will perform for various tasks (language modelling, classification, etc.) as with all modelling. There are generally two ways for the evaluation of topic models i) extrinsic methods [61] and ii) intrinsic methods [35]. A common evaluation method for topic models is the probability of held-out documents or perplexity, an intrinsic approach to evaluating the quality of topic models. This approach uses the language model framework to dictate how well the topic model performs. Though commonly used, it is reported in [35] that the perplexity is not always the best predictor for how well topic models results are against human judgment. [10] reports that topic models that achieve better predictive perplexity often have less interpretable results, they fail against human judgement. Although [38] mentions that perplexity is useful for model selection and adjusting of parameters. Word intrusion and topic intrusion are evaluation measures that explicitly evaluate the quality of the topics inferred and how well the model assigns topics to documents. A different evaluation method discussed in [35] suggests a coherence measure that corresponds well with human judgment and makes it possible to identify problems with topic models without human or external intervention.

2.4.2 Distance Metrics

We need to measure how close two documents are for applications is retrieval, where similar documents that contain similar information may be regarded as relevant even if one may contain the query words while the other does not. The distance between documents allows for us to organise information and as a consequence we are able to retrieve relevant information with high accuracy.

With a quantitative data this can be done with various distance metrics. With qualitative data, such as text, it proves to be challenging. 'Closeness' in this regard means both **lexical similarity** and **semantic similarity**. In our study we consider the following measures:

- Cosine.
- Soft cosine.
- Jensen-Shannon divergence.

Cosine Measure for Similarity

The cosine similarity measure is one of the most widely used measures for similarity between term vectors [21,27]. It measures the cosine of the angle between two vectors projected on a multi-dimensional space. It is highly effective for sparse terms vectors as only non-zero dimensions need to be considered [27]. We discuss this measure further in section 3.3.1.

Soft Cosine Similarity

The cosine measure of similarity is widely applied and is normally taken for granted [53]. It has been proposed in [27, 32, 53] that the cosine similarity be modified, as it has implicit biases in its calculation of similarity as it assumes there is no similarity between feature, for example the word 'car' and 'drive' are different, but are similar in their use and meaning in certain contexts. Thus in [53] a measure of similarity that takes into account feature similarity is proposed [53]. When feature similarity is considered for the cosine similarity measure, this is what is called **soft cosine similarity**.

Jensen-Shannon divergence

The Jensen-Shannon divergence measures the distance between two or more continuous or discrete probability distributions [28]. It is closely related to other divergence measures such at the Kullback-Leibler divergence and mutual information measure.We use this measure to determining similarity between two distribution for the LDA model. We discuss this measure further in section 3.3.3. In the next section we discuss the Vector Space Model Framework and where the similarity measures described above fall in it.

Chapter 3

Vector Space Models

Computers have little understanding of natural language, whether in text or speech format. This poses a problem for tasks such as information retrieval and the application of machine learning algorithms. The use of vector space models to transform text documents to some organised structure that can be analysed by models requires the use of feature extraction methods, which sometimes referred to as *vectorisation*. For text analysis, we have full documents or sentences that may differ in length from quotes or tweets to whole books, but whose vectors are always of a consistent size [26]. We define terminology and notation in Table 5.1 which will be used in the rest of the chapter.

Table 3.1: Table of definitions

Term	Definition
$\mathbf{D}$	Entire document collection or the corpus
$\mathbf{X}$	A term-document matrix that captures the frequency of term i in document j
M	Number of documents in a corpus. A corpus is a collection of documents. A document is a sequence of words and $d \in \{d_1, d_2, ..., d_n\}$
V	Vocabulary size. The vocabulary is a set of unique words that are present in the corpus
d_j	Document j in collection of Documents $\mathbf{D}$
w_i	The i-ith word in vocabulary V: $w_i \in \{w_1, ..., w_V\}$

For a collection of documents $\mathbf{D} = (d_1, d_2, \ldots, d_M)$, such as blog posts, tweets and news paper article, we must find a way to represent each document d_j for $j \in \{1, 2, \ldots, M\}$ in such a way that we are able to describe each document with a fixed vocabulary of words (w_i with $i \in \{1, 2, \ldots, V\}$) [51]. Each document is then a sequence of words $d_j = (w_1, w_2, \ldots, w_V)$ in the vocabulary and is represented by a fixed V-dimensional vector. This process results in the generation of the *term-document* matrix $\mathbf{X}$, which in some text is referred to as the *word-document* matrix. In this structured form text is represented properly for machine tasks, e.g. information retrieval [40]. We have described the vector space framework and the bag-of-words model is a prominent example of this framework due to its simplicity and has been used to encode semantic space for our study as well. With the vector space framework of text we are forced to think of documents as points in a multidimensional space and can measure how close or far a document from a collection is to another document not in the collection, referred to as a *query*.

A query can be represented as a vector, where term q_{ij} $(0 \leq i \leq V, 0 \leq j \leq M)$ is a non-negative value denoting the number of occurrences of term j in query i. Both the document vectors and the query vector provide the locations of the objects in the semantic space. By computing the distance between the query and other objects in the space, objects semantically similar to the content in the query will be retrieved. There are various vector representations like one-hot vector encoding, bag-of-words and the *terms-frequency inverse document frequency* or TF-IDF approach. We discuss the bag-of-words in the next section.

3.1 Bag of Words

A bag of word (BoW) model is a vector space model that captures the frequency of the word occurrence $w_i \in V$, V is some vocabulary set for some document $d_j \in \mathbf{D}$, and $\mathbf{D}$ is a collection of documents or a corpus. Here we make the simple assumption that each one $w_i \in V$ is sampled independently from one another from some discrete distribution [37]. It is important to note that information relating to order and structure is lost due to this method of feature extraction. In information retrieval, the BoW representation assumes we can estimate the relevance of documents to a query by representing the documents and the query as bags of words [57]. The BoW representation is the simplest encoding of a semantic space, whose primary insight is that meaning and similarity are captured in vocabulary [56].

	word1	word2	word3	...	word*n*
doc1	11	5	1		1
doc2	0	1	2		8
⋮					
doc*n*	3	2	0	...	9

Figure 3.1: Bag-of-words representation

For a set of M documents and a vocabulary of size $|V|$ then the bag of representation is illustrated in Figure 3.1. This approach is used in our study of relevance due to the simple framework it is defined on and easy implementation.

3.2 Term Frequency Inverse Document Frequency

Term frequency inverse document frequency (*tf-idf*) is based on BoW, but provides more detail [6, 7]. In information theory, a rare event provides more information than an expected event [57], a way to formalize this is the *tf-idf* weighting scheme: This weighting scheme shows how important a given word is, not only by looking at the term frequency, but also analyzing how many times the word occurs across documents and has shown significant improvement over raw frequency [57]. The *tf-idf* also handles length normalisation, since search engines tend to have biases in favour of longer documents [57]. We break down the function to better appreciate its result. To discriminate between documents for the purpose of scoring the use of a document-level statistic, such as the number of documents containing the term is far better than a collection wide statistic for the term [52]. The inverse term document is defined as:

$$idf_t = log(\frac{M}{df_t}),$$

M denotes the total number of documents in our collection and df_t is the document frequency for term t. This ensures the *idf* of rare terms is high and the *idf* of frequent terms low. This leads to the mathematical formulation of *tf-idf*:

$$\text{tf-df}_{t,d} = tf_{t,d} \times idf_{t,d}.$$

We observe that a term t that occurs frequently in a small number of documents receives a higher weighting, this is through the fact that $tf_{t,d}$ for term t will be some high number and $idf_{t,d}$ will also be high, due to the fact that $\frac{M}{df_t}$ will be some number greater than 1, therefore $log(\frac{M}{df_t})$ will also some number bigger than zero. But for $\frac{M}{df_t}$ larger than 6, we will have a idf_t score greater than 1, which translates to a higher *tf-idf* score for this rare term t. This scheme then assigns more weight and thus more consideration when determining the relevance of a document. Therefore the terms that occur frequently in a document and across documents, receive less consideration when it comes to the task of relevancy. Consider the following example :

'On this document I will write this symphony'
'On this I will write that'
'On this I fear I will not be able to write this nor that'

The term frequencies of each term in the vocabulary across all documents is represented in Table 3.2 and the *tf-idf* scores for terms in Table 3.3.

Term	Document 1	Document 2	Document 3	Document Frequency
on	1	1	1	3
this	2	2	1	3
document	1	0	0	1
i	1	2	1	3
will	1	1	1	3
write	1	1	1	3
symphony	1	0	0	1
that	0	1	1	2
fear	0	0	1	1
not	0	0	1	1
be	0	0	1	1
able	0	0	1	1
to	0	0	1	1
nor	0	0	1	1

Table 3.2: The table contains the frequency of each word in each document.

Term	Document1	Document2	Document 3
that	0	0.40546511	0.40546511
document	1.09861229	0	0
symphony	1.09861229	0	0
I	0	0	0
fear	0	0	1.09861229
to	0	0	1.09861229
this	0	0	0
nor	0	0	1.09861229
be	0	0	1.09861229
not	0	0	1.09861229
write	0	0	0
will	0	0	0
On	0	0	0
able	0	0	1.09861229

Table 3.3: The table contains *tf-ifd* transformation of each word in each document.

In table 3.2 the documents are converted into a bag of words representation, where each words frequency in each document is the only recorded attribute, the document frequency is also recorded. Then we calculate the *tf-idf* score of each term in all document, words like "on" and "this" that were recorded to appear in all three documents are affected by the $idf = log(\frac{M}{df})$ factor of the score, where M is the number of documents in the corpus. The closer the df_t for term t the fraction is closer to one, making the log of the expression converge to 0. This factor allows rare words that occur in a few documents in the corpus to get a high scores where as words that occur with low high frequency across the corpus receive scores closer to zero. We interpret the score to mean that the closer a term's TF IDF score is to 1, the more informative the term is to it. The nearer the score is to zero, the less the word is informative [7].

3.3 Similarity

Once the corpus is vectorised into a vector space, we are able to perform mathematical calculations and modelling. Certainly one of the most common calculation on vector spaces is that of similarity: To be able to determine the relatedness between a collections of documents and a query. In this section we discuss similarity measures in the VSM framework. We discuss the methods on to measure similarity in the vector space framework.

3.3.1 Cosine Similarity

When documents are represented as term vectors, the measure of similarity between them corresponds to their correlation [21]. This correlation is quantified by the cosine of the angle between these vectors and thus cosine similarity is defined as the dot product

$$a \cdot b = \sum_{i=1}^{N} a_i b_i,$$

and the norm is defined as

$$||x|| = \sqrt{x \cdot x}.$$

Then the cosine is defined as

$$\text{cosine}(a, b) = \frac{a \cdot b}{||a|| \cdot ||b||},$$

which can be written as

$$\text{cosine}(a, b) = \frac{\sum_{i=1}^{N} a_i b_i}{\sqrt{\sum_{i=1}^{N} a_i^2}\sqrt{\sum_{j=1}^{N} b_j^2}}.$$

This measure represents how two documents are correlated and is bounded in the closed set $[-1, 1]$, where -1 means that the two documents are opposed to each other, 0 is interpreted as the two documents are not similar to one another and 1 suggests that the two document are perfectly correlated, which translates to the documents being identical. In the case of IR the cosine similarity will remain between 0 and 1 since we deal with positive valued vectors.

3.3.2 Soft Cosine Similarity

Cosine similarity is a common measure to compare similarity between two vectors, however some of the assumptions made for this measure do not apply in the NLP space. In [53] a modified version of the cosine similarity is proposed, since the cosine similarity is overly biased by features with higher values and does not care much about how many features are shared by two vectors [31]. Their proposal is a soft cosine measure, which takes into account the similarity between features and as a result relaxes the assumption of independence between features. Consider the basis vector representation of documents as:

$$\begin{aligned} e_1 &= (1, 0, \ldots, 0) \\ e_2 &= (0, 1, \ldots, 0) \\ &\ldots \\ e_{|V|} &= (0, 0, \ldots, 1), \end{aligned}$$

This representation for one word documents or representation of a single feature in the VSM assumes that words are independent. But this notion is untrue eg:

'*game*' and '*play*' are different words and may be represented that way in the VSM, but they are similar in terms of meaning. As a result we have that:

$$cosine(e_i, e_j) = 0$$

The assumption made in [53] is that similarity can be modelled using cosine between features:

$$cosine(e_i, e_j) = s_{ij} = sim(f_i, f_j),$$

where f_i and f_j are features corresponding to the basis vectors and $sim(\cdot)$ is a similarity measure such as synonymy. Soft similarity is defined as :

$$softcosine(a, b) = \frac{\sum\sum_{i=1}^{N} s_{ij} a_i b_j}{\sqrt{\sum\sum_{i=1}^{N} s_{ij} a_i \cdot a_j}\sqrt{\sum\sum_{j=1}^{N} s_{ij} b_i \cdot b_j}},$$

where $s_{ij} = sim(f_i, f_j)$ and if there is not similarity between f_i and f_j, $s_{ii} = 1$ and $s_{ij} = 0$ when $i \neq j$. This is done using the **Levenshtein** distance - a string metric for measuring the difference between two sequences, which is suitable for NLP tasks, as they deal with text. In their study they consider words, n-grams and syntactic n-grams as their features. We only consider words as features in this paper.

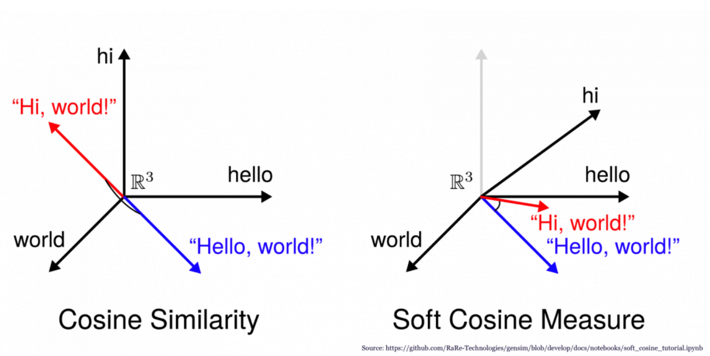


Figure 3.2: Illustration of cosine similarity [1]

This similarity measure that considers similarity based on semantic closeness is required and the normal cosine similarity measure assumes terms in a vector space are independent, regardless of whether they belong to the same topic. This is where the soft-cosine measure takes into account similarity of features from the same topic, this is illustrated in 3.2.

3.3.3 Jensen-Shannon Divergence

The Jensen-Shannon divergence is a method of measuring similarity between probability distributions. It is closely related to the Kullback-Leibler divergence but is symmetric. The square root of this measure is a metric and is called the Jensen-Shannon distance. We state the more mathematical definition of this measure. Consider the set $P(E)$ of probability distributions where E is a set provided with some σ-*algebra* of measurable subsets. The Jensen-Shannon divergence $J(D||Q)$: $P(E) \times P(E) \rightarrow [0, \infty)$ is defined as

$$JSD(P||Q) = \frac{1}{2}D(P||M) + D(Q||M),$$

where $M = \frac{1}{2}(P+Q)$ and $D(\cdot||\cdot)$ is the Kullback-Leibler divergence measure. This is a symmetric and smoothed version of the Kullback-Leibler divergence. For a more generic version allowing for more than one comparison we let $\pi_1, \pi_2, ..\pi_n$ where $\pi_i \geq 0$ for $i \in (1, 2, ..n)$ and $\pi_1 + \pi_2 + .. + \pi_n = 1$, be weights for n probability distributions then we define the $JSD(\cdot||\cdot)$ to be :

$$JSD_{\pi_1,\pi_2,...,\pi_n}(P_1, P_2, ..., P_n) = H(\sum_{i=1}^{n} \pi_i P_i) - \sum_{i=1}^{n} \pi_i H(P_i), \tag{3.1}$$

where $H(P)$ is the Shannon entropy and where $\pi_1, \pi_2 \ldots, \pi_n$ are weights that are selected for the probability distributions $P_1, P_2, \ldots, P_n$.

3.3.4 Conclusion

We have formed a basis for text representation for machine learning algorithms in the form of the bag-of-words models and have discussed the assumptions associated with it. We also discussed measures to measures similarity for the vector space model framework. This lead us to the discussion of the Jensen-Shannon measure of divergence since topic models are vectors of distributions. We also discussed the soft cosine measure for when the assumption of independence can be replaced by some other similarity measure. Since we have established a basis for machine learning in text, in the next chapter we discussed supervised approaches to text categorisation in the form of the naive Bayes classifier.

Chapter 4

Supervised Text Classification

Text classification is a machine learning problem found in a variety of fields, such as email spam detection, due to the need for personal organisation [43]. Classification is commonly addressed by **supervised learning** approaches. In supervised learning we are given an input x and a fixed set of M classes, $Y = \{y_1, y_2, \ldots, y_M\}$, and we are tasked with predicting a class $y \in Y$ [23]. For the supervised learning environment there is a set of manually labelled training data and we want a method to accurately classify new, previously unseen documents. There are various classifications techniques such as logistic regression, decision trees and support vector machines (SVM). We focus on the naive Bayes classifier as a procedure to calculate probabilities, as it provides simple implementation due its assumptions and desirable performance results.

4.1 Introduction

The ability to learn a concept from a few examples is one of the core capacities of the human mind. An example of concept of learning is when a child learns what a dog is, they can accurately identify a dog after one positive example. When a cat is incorrectly identified as dog, corrections provides clarification on the dog concept. Therefore negative examples are useful, but more to refine the concept rather than to learn it [55]. For machines, concepts are learned through features extracted from data, then some function distinguishes what belongs to a concept and what does not based on the collected features. Feature selection is influenced by a practitioner's bias such as, the use of stopword for certain natural language processing applications, this results in a bias in results. In supervised learning this collection of methods are called *classifiers*[1].

[1] A classifier is defined by a deterministic function that assigns a label c for each example $\mathbf{x}$ it is given

A probabilistic classifier can attach a probability to an observation belonging to some class c and this is useful for decision making [23]. In supervised learning there are two approaches to classification i) **generative classifiers** and ii) **discriminative classifier**. A generative classifier models on the assumption that it can accurately model the input data and can predict the corresponding class as a result where as discriminative classifiers learn what features from the input data are most informative to be able to separate classes. The discussion forms a basis for the next section.

4.2 Naive Bayes

The naive Bayes is a probabilistic classifier, which means that given some documents d_j, classification will be based on the maximum posterior probability given document d_j. We represent this estimation of the correct class with $\hat{c}$ using Bayes theorem which is defined as follows [23]:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)}. \tag{4.1}$$

We are able to infer the class of the document d_j by transforming (4.1) into the following expression:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d_j) = \underset{c \in C}{\operatorname{argmax}} \frac{P(d_j|c)P(c)}{P(d_j)}. \tag{4.2}$$

We calculate the most probable class $\hat{c}$ by finding the product of the prior probability of c and the likelihood function $P(d_j \mid c)$. The denominator of equation 4.2 can be thought of as a normalising constant and can be ignored to arrive to the following equation:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(d_j|c)P(c). \tag{4.3}$$

4.2.1 Assumptions

Consider a document $d_j = (w_1, w_2, \ldots, w_{|V|})$, with a fixed length of size $|V|$, and V is the vocabulary for some corpus $\mathbf{D}$. We consider equation 4.3 with the consideration of d_j to get to the following:

$$P(d_j \mid c) = P(w_1, w_2, \ldots, w_{|V|} \mid c) \tag{4.4}$$

equation 4.4 proves to be difficult to calculate, since it requires the estimation of all possible combinations of the words and take into account order. We make assumptions to simplify the modeling constraints that have been discussed. The first assumption is the bag of words assumption (as discussed in Chapter 3) as the vector representation of the documents. We assume that order of the words does not matter, only how many times it occurred in the document. The second assumption is that each w_i is independent on the condition we are given the class label c. We have that equation 4.4 becomes :

$$P(d_j|c) = \prod_{i=1}^{|V|} P(w_i|c). \tag{4.5}$$

And equation 4.3 is simplifies into the following:

$$\hat{c} = \underset{c \in C}{\text{argmax}} \prod_{i=1}^{|V|} P(w_i|c)P(c). \tag{4.6}$$

The final expression is derived from apply the *log*-function to equation 4.6. This transformation speeds up the modelling and caters for underflow[2]. As a result of the reasons aforementioned we get the following result:

$$\hat{c} = \underset{c \in C}{\text{argmax}}\, logP(c) + \sum_{i=1}^{|V|} P(w_i|c). \tag{4.7}$$

4.2.2 Estimation

We have to calculate the probabilities $P(c)$ and $P(w_i \mid c)$ from our given data, this is also called training the models. We estimate parameters and the algorithm using these estimated parameters to classify new documents with equation 4.8. The parameters of an individual class follow a multinomial distributions over words and are the collection of probabilities for a given c, $\theta_{w_i|c} = P(w_i|c, \theta)$. We then need to only attach parameter to the weights, θ_c , where $\theta_c = P(c|\theta)$. We revise equation 4.8 to include the parameters we have introduced:

$$\hat{c} = \underset{c \in C}{\text{argmax}}\, logP(c|\hat{\theta}_c) + \sum_{i=1}^{|V|} P(w_i|c, \hat{\theta}_{w_i|c}) \tag{4.8}$$

We see that θ is a set of multinomial with prior probabilities over those multinomials i.e.:

$$\theta = \{\theta_{w_i|c} : w_i \in V; \theta_c, c \in C\}$$

The estimate $\hat{\theta}_{w_i|c}$ is the number of times word w_i appears in c for the training set over the total number of word occurrences in the the training set which belong to the class c. And we represent that estimate to be:

$$\hat{\theta}_{w_i|c} = \frac{1 + N_{w_i,c}}{|V| + N_c} \tag{4.9}$$

where $N_{w_i,c}$ is the number of times word w_i occurs in class c. N_c is the total number of words in class c and $|V|$ is the size of our vocabulary. Notice that

[2]Underflow is a condition in a computer program where the result of a calculation is a number of smaller absolute value than the computer can actually represent in memory on its CPU.

there is a 1 added to the numerator value in equation 4.9, this is called *Laplace Smoothing*. We consider a scenarios a certain word may not occur in certain class in our training set. Then for that word w_i we will have $\hat{\theta}_{w_i|c}$ equal to zero, subsequently this will turn into a zero likelihood considering the conditional independence assumption made for the naive Bayes. We must then replace this zero probability, with a small non-zero probability in the form of Laplace smoothing. There are various other smoothing methods we do not consider our study. Then we place our focus on the prior probability for c. This is estimated to be the proportion of documents that have class c over all documents in the training set:

$$\hat{\theta}_{c_j} = \frac{N_c}{N_{\mathbf{D}}} \tag{4.10}$$

where N_c is the total number of words in class c and $N_{\mathbf{D}}$ is the total number of documents. We use the results we acquired from training the model and the Bayes Theorem to get the result above. To classify a document into a class c we simply find the values of $\hat{c}$ in equation 4.8 that is a maximum across all $\hat{\theta}$ and the corresponding class label is chosen. In algorithm 1 the process of estimating $\hat{\theta}_{w_i|c}$ and $\hat{\theta}_c$ is described. We follow this illustration with an example of how

Algorithm 1 Algorithm for Naive Bayes Classifier

$N_c = 0$, $N_{w_i,c} = 0$
$D = d_i$
for $j = 1 : C$ **do**
  $c = y_i$// `the` i^{th} `example's label`
  $N_c := N_c + 1$
  for $t = 1 : |V|$ **do**
    if $w_t = 1$ **then**
      $N_{w_i,c} := N_{w_i,c} + 1$
    end if
  end for
end for
return $\hat{\theta}_{w_i|c} = \frac{1+N_{w_i,c}}{|V|+N_c}$, $\hat{\theta}_c = \frac{N_c}{N_{\mathbf{D}}}$

the algorithm works.

Example

To illustrate what we have discussed we look at a simple example of classifying text into two categories; sport and non sport. Given the training data Table 4.1, we calculate the posterior of each of the classes given a test document d_{test}.

Index	Document	Label
1	a great game	sports
2	the election was over	non sports
3	very clean match	sports
4	a clean but forgettable game	sports
5	It was a close election	non sports

Table 4.1: Multinomial naive Bayes on sports text classification

A basic probability calculation for each word in both classes is done to get the predictions for the test document.

$$d_{test} = a\ very\ close\ game$$

The Naive Bayes gives us the following scores

$P(sport \mid d_{test})$	$P(non\_sport \mid d_{test})$
2.7648 e-05	5.7175e-06

We then decide to classify this document in class sports category. In our example we have that the posterior probabilities for each class are small quantities. This is as a result of the naive Bayes assumption of of conditional independence for each feature given the class. For very large vocabularies we find that these probabilities become so small they cannot be represented by some computer programs. This is called numerical underflow and it results in false results being presented for products of small numbers. A way to solve this is to use the *log* and *e* functions since $log e^a = a$. This combined with one of the rules of the logarithm which is:

$$log(\prod_{i=1}^{N} x_i) = \sum_{i=1}^{N} log(x_i) \tag{4.11}$$

Note that even with this provision numerical underflow is still a possibility. We have that any complex multiplication scheme with small numbers becomes a summation computation and we avoid underflow. This is called the **log-sum-exp** trick and it is discussed and derived in [37].

4.3 Applications

We train the multinomial naive Bayes classifier on two datasets namely the 20Newsgroups and the fake news data set, we discuss the data sets in Chapter 6. We use scikit learn version 0.20.1 library in Python 3.7 for the multinomial naive Bayes and the support vector machine classifiers. We break the data up into a training and test set, to extract feature and get prediction performance respectively. We begin our application with the 20Newsgroups data and choose 5 categories, three that are different to each other (religion, politics and computer hardware) and two that are similar (sports-motorcycle and sports-auto). We would like to test how well the classifier can distinguish between classes that are different and those that are potentially similar. For the multinomial naive Bayes we observe the following:

Class	Precision	Recall	F-1 score
rec.motorcycles	0.93	0.96	0.95
talk.religion.misc	0.93	0.92	0.92
talk.politics.mideast	0.95	0.95	0.95
rec.autos	0.99	0.97	0.98
comp.sys.ibm.pc.hardware	0.96	0.94	0.95

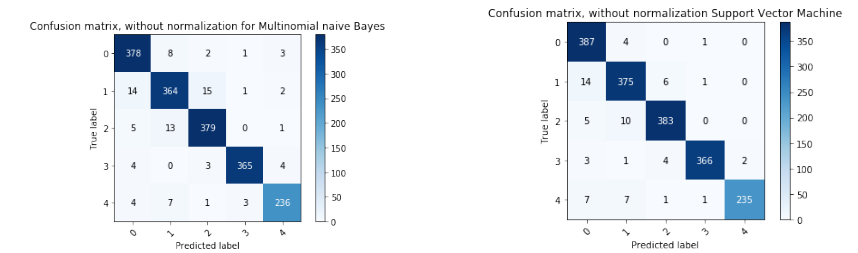
Table 4.2: Classification report for naive multinomial on 20Newsgroups

We look at the classification report for the prediction performance of the multinomial classifier, with smoothing parameter, *alpha*, set to 0.01, the class probabilities collected from our training data and we do not set the prior probabilities for the classifier. We train the classifier on 590 documents from the motorcycle class, 594 documents from the religion class, 598 from the politics class, 564 from the auto class , and 377 from the computer hardware class. We can see that there is class imbalance, and the difference is stark between the computer class and the rest of the other classes. Overall all the naive Bayes classifier accuracy to identify positive examples in each class, the precision, is above 93% for all classes or 93 out of 100 positive examples are true positives and the rest are false positive. We also look at recall as measure of misclassification, here we measure the number of true positives identified by the classifier against the total number of positive examples present in our testing, those that were identified and those that were misclassified as negatives. The recall for the naive Bayes is also above 92% for all classes. This means on average the classifier per class will correctly identify above 90% of the examples presented to it it with the features it has selected, which is measured using the F1-score, a harmonic mean between the recall and precision. We look at the support vector machine classifier results in comparison to the naive Bayes classifier.

Class	Precision	Recall	F-1 score
rec.motorcycles	0.93	0.99	0.96
talk.religion.misc	0.94	0.95	0.95
talk.politics.mideast	0.97	0.96	0.97
rec.autos	0.99	0.97	0.98
comp.sys.ibm.pc.hardware	0.99	0.94	0.96

Table 4.3: Classification report for support vector machine on 20Newsgroups

For the support vector machine classifier we notice better precision and recall results for the region category with 2% more precision and 3% better recall than the naive Bayes classifier. This results in 1% overall better per-



(a) Multinomial naive Bayes classifier (b) Support vector machine classifier

Figure 4.1: Confusion matrix analysis

formance by the vanilla support vector machine where norm-L2 is used in the penalisation and the squared hinge loss function is used to calculate loss. Please find default parameterisation for the svm classifier in scikit-learn[3]. We also take a look at the confusion matrices for the two classifiers. Where 0-rec.motorcycles, 1-talk.religion.misc, 2-talk.politics.mideast, 3-rec.autos and 4-comp.sys.ibm.pc.hardware.

For the fake news data we collect 27812 features from 16640 training documents to arrive to the following results for both the multinomial naive Bayes classifier and support vector machine classifier in Table 4.4 and Table 4.5 respectively.

Label	Precision	Recall	F-1 score
Reliable	0.89	0.93	0.91
Unreliable	0.93	0.88	0.90

Table 4.4: Classification results fake news classifier multinomial naive Bayes classifier

From Table 4.4 we observe that for the naive Bayes classifier has a high precision score between both class achieving a minimum of 89% on the reliable class, this means 89% of the documents identified as reliable or unreliable where correctly identified and only 11% were misclassified. Though the classifier does better identifying fake documents than real ones. We compare these results to those of the SVM or support vector machine.

Label	Precision	Recall	F-1 score
Reliable	0.96	0.93	0.95
Unreliable	0.94	0.96	0.95

Table 4.5: Classification results fake news classifier support vector machine

[3]https://github.com/scikit-learn/scikit-learn/blob/95d4f0841/sklearn/svm/$_{c}lasses.pyL13$

We observe that the SVM classifier does markedly better when compared to the naive Bayes on the task of classifying documents from the fake news data set. Using the F1-score we observe that out of 100 examples the SVM classifier will on average classify more reliable and unreliable documents than the naive Bayes. We look at the Receiver Operating Characteristic curve or ROC curve of both the classifier to gather more insight on their performance.

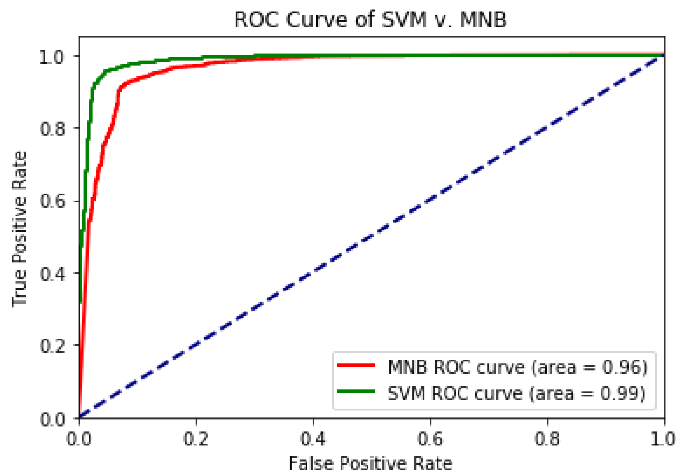
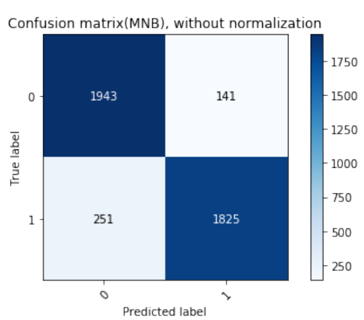
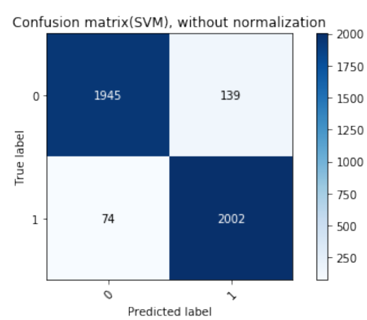


Figure 4.2: ROC Curve: support vector machine against naive Bayes

In Figure 4.2 we observe that the SVM is more accurate than the multinomial naive Bayes classifier and the accuracy on the test documents reflects this with the SVM sitting with 99%accuracy while the naive Bayes achieves a 96% accuracy on the selected test documents. We take at the confusion matrices next to inspect the misclassification errors made by each classifier.



(a) Multinomial naive Bayes classifier



(b) Support vector machine classifier

Figure 4.3: Confusion matrix analysis

We have that in Figure 4.3 the label 0 corresponds to the Reliable class and the label 0 corresponds to the Unreliable class in fake news data. Looking at the heat-map representation of each of the matrices we can identify that the SVM suffers less from misclassification between classes. When the document is from an unreliable source the naive Bayes has more confusion with 251 cases misclassified as a result, while the SVM does better with 74 misclassified cases. Aside from this difference test can be conducted to test if the misclassfication is statistically significant. We discuss the conclusion of the application in the conclusion section of this chapter. We discuss a more complex version on the naive Bayes in the next section.

4.3.1 Bayesian Naive Bayes

With the naive Bayes classifier, classification occurs after we have estimated our θ_c and θ_{jc} , which are the class probability and feature vector j's probability given class c. From text it has been discussed that the maximum likelihood can over-fit ($\hat{\theta_c}$, $\hat{\theta_{jc}}$). This leads to cases where the naive Bayes classifier can fail. To combat this issue a fully Bayesian approach is taken.

4.4 Conclusions

In this chapter, we introduced the naive Bayes classifier as an important building block in our understanding of latent semantic representations for text for two reasons: Firstly, it makes use of the bag-of-words vectorisation which takes into account word frequencies as features. Secondly, it is a generative classifier which makes assumptions on how the data was generated. In our application we also discover that even though the naive Bayes vanilla classifier performs worse when compared to the support vector machine, it still achieves high classification performance results. This is advantageous in the case of dealing with large data sets where the classifiers simple assumptions will result in cheaper computation and parameter tuning. The the next chapter, we investigate unsupervised text models which follows the same generative assumption than the naive Bayes, namely that the words are generated from a multinomial distribution.

Chapter 5

Latent Variable Models for Discrete Data

In the previous chapter we introduced the Naive Bayes, a supervised approach to text analysis. The volumes of text being generated daily has created the need for unsupervised methods in information retrieval. In this chapter we give an overview of two unsupervised text analysis methods.

Table 5.1: Table of definitions

Term	Definition
α	Hyper parameter of prior distribution.
X	Random variable that represents the observed data.
Z	Latent variable, unobserved.
N_d	Size or length of document d.
β	Word-topic parameter.
w_i	The i-ith word in vocabulary V: $w_i \in \{w_1, ..., w_V\}$.
θ	Parameter of prior distribution.
d	Documents in a collection.

5.1 Latent Semantic Indexing

Human-computer interaction is by means of natural language queries - the user submits a query, by providing keywords or some free form text [19]. The Latent Semantic Indexing (LSI) model was designed to address the challenge of matching words in a query with those in the collection of documents to be searched. The rationale behind this method is to map terms and documents on to the same space, to create some *latent semantic space*, where documents that share co-occurrence counts will have similar representations in the latent semantic space even if they have no terms in common [13, 19]. This method relies on the Singular Value Decomposition (SVD) method to perform dimension reduction on the *document* $\times$*term* matrix. Using matrix representation we illustrate the SVD method for the LSI in figure 5.1. LSI has been shown to address two challenging NLP issues, namely *polysemy* and *synonymy* by taking advantage of higher order structures in the association of terms with documents in order to improve the detection of relevant documents on the basis of terms found in a query [18]. One advantage of the LSI is that the decomposition provides an orthonormal basis which is computationally convenient because one decomposition for T dimensions will simultaneously give all lower level dimensional approximations as well [18].

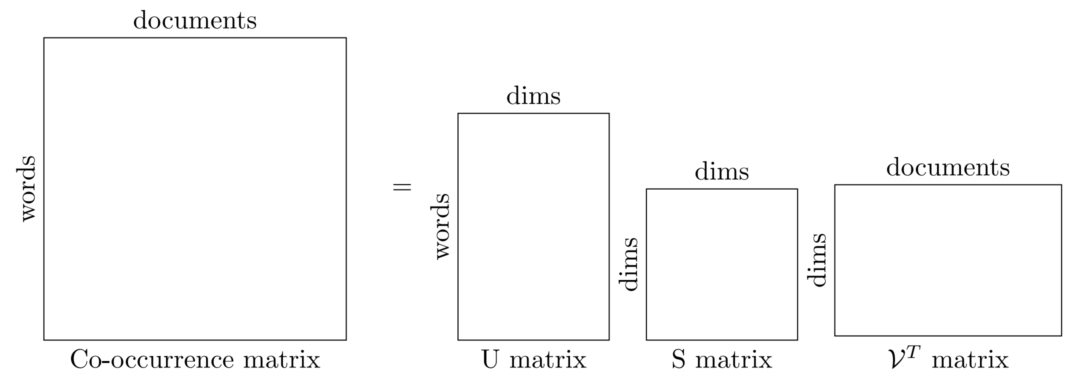


Figure 5.1: Matrix representation of LSI

5.1.1 Probabilistic latent semantic index model

The LSI has been applied with remarkable success in different domains but it has a lot of deficits, mainly due to its unsatisfactory statistical foundation [19]. Hofmann(1998) then suggested the probabilistic Latent Semantic indexing model (pLSI), a novel approach to automated document indexing which is based on a latent class model for factor analysis of count data. This model has a solid statistical foundation since it is based on the likelihood principle and defines a proper generative model for data [19]. The core of the pLSI is model called the *aspect model*, a latent variable model for general co-occurrence data which associates a latent variable $z \in \{z_1, z_2, ..z_K\}$ with each observation of the word $w \in \{w_1, w_2, ..w_{N_d}\}$ in document $d \in \{d_1, d_2, ..d_N\}$. To generate a document-word pair, we select a document d with probability $P(d)$, then we select a latent variable z with probability $P(z|d)$ then we are able to generate a word w with probability $p(w|z)$. The joint probability distribution between word w

and document d is given by:

$$\begin{aligned} P(d,w) &= P(d) \cdot P(w|d) \\ &= P(d) \cdot \sum_z P(w|z) \cdot P(z|d) \\ &= P(d) \cdot \sum_z P(w|z) \cdot \frac{P(d|z) \cdot P(z)}{P(d)} \\ &= \sum_z P(w|z) \cdot P(d|z) \cdot P(z) \end{aligned}$$

We then have that the probability of generating a new document of length N_d as a bag of word is:

$$P(w_1, w_2, ..w_{N_d}) = \prod_i^{N_d} \sum_{z=1}^{K} P(w_i|z) \cdot P(z|d)$$

We look at Figure 5.2 to see a graphical representation of the document generation process. The shaded circles in the plate model are observed variable, where as circles with a white background are unobserved.

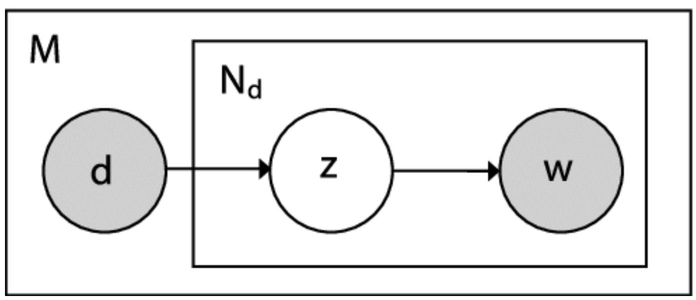


Figure 5.2: pLSI graphical model

Though the effectiveness of this model was shown to be higher than that of term-matching and the LSI technique, the effectiveness of mixture models on IR is not yet established. The pLSI has a problem in that the generative semantics are not well defined [61] .

5.2 Latent Dirichlet Allocation

A generative model describes how data is generated in terms of a probabilistic model. Because generative models make assumptions about how the data (documents in this case) is generated, it allows for sampling from the aforementioned distributions. The Latent Dirichlet Allocation (LDA) model [9] is a generative model which makes assumptions about how documents in a corpus are generated and produces estimates for *topic* $\times$ *word* and *document* $\times$ *topic* distributions [12]. The goal of LDA is to find sparse representations of the members of a collection that enable efficient processing of large collections while preserving the essential statistical relationships that are useful for task such novelty detection, similarity and relevance judgment [9].

The output of LDA is a finite index of hidden topics which describe the underlying documents. LDA is a hierarchical Bayesian model – the hierarchical part comes from the fact that the generative process is assumed to be broken up into three levels. The first level being the word level, the next being the an abstract concept level like a topic and the last being the document level. The LDA assumes the following generative process for each document d_i in the corpus D :

1. Choose $N|\eta \sim POISSON(\eta)$.
2. Choose $\theta|\alpha \sim Dir(\alpha)$.
3. For $n \in \{0, 1, 2, 3, ..N\}$.
   (a) Choose topic $Z_n|\theta \sim Mult(\theta)$.
   (b) Choose word $W_n|\{z_n, \beta_{1:n}\} \sim Mult(\beta_{z_n})$.

LDA makes the following assumptions:

- Dimensionality of Dirichlet distributions is assumed to fixed and known.
- Word probabilities are treated as fixed quantities that will be estimated.
- The Poisson assumptions is not critical and more realistic document length can be used.

We note that N is independent of θ and Z.

5.3 Learning algorithms

A central task in the application of probabilistic models is the evaluation of the posterior distribution of the latent variable and the evaluation of the expectation computed with respect to this distribution [8]. For many models of interest the posterior distribution is intractable, this is due to factors such as dimensionality of the latent hidden variables and in other cases the problem arises from the marginal distribution of the observed data X. Two schools of thoughts currently exist for approximation: Sampling and optimisation. In this section, we illustrate a sampling method (Gibbs sampling) and an optimisation method (Variational inference) which are appropriate for LDA parameter estimation.

5.3.1 Gibbs Sampling

Markov Chain Monte Carlo (MCMC) refers to a set of approximate iterative techniques designed to sample values from complex distributions. Gibbs sampling [17] also known as alternating conditional sampling is a specific form of MCMC and simulates high dimensional distributions by sampling lower dimensional subsets of variables, where each subset is conditioned on the value of all others. The sampling is done sequentially and proceed until the sampled values approximate the target distribution [18] the target distribution normally referring to the posterior distribution for approximate inference tasks [63]. Murphy [37] describes it as the MCMC analog of coordinate decent. We adopt Zeger's(1991) explanation of the Gibbs sampling process. Assume there are three random variables U,V and W of interest. Let $P(U|V,W)$, $P(V|U,W)$ and $P(W|U,V)$ denote conditional distributions that possess a simpler form when compared to the joint distribution denoted by $P(U,V,W)$. We let the joint distribution be fully determined by the conditional distribution. The aim is to generate random variate from U,V and W as follows: It has been shown that

Algorithm 2 Gibbs Sampler

initialize $U^{(0)}$, $V^{(0)}$ and $W^{(0)}$
for $j = 1 : B$ **do**
    `Draw` $U^{(j)} \sim P(U|V^{(j-1)}, W^{(j-1)})$
    `Draw` $V^{(j)} \sim P(V|U^{(j)}, W^{(j-1)})$
    `Draw` $W^{(j)} \sim P(W|U^{(j)}, V^{(j)})$
end for
return $(U^{(B)}, V^{(B)}, W^{(B)})$

as $B \to \infty$, the joint distribution $(U^{(B)}, V^{(B)}, W^{(B)})$ converges to $P(U,V,W)$ at an exponential rate [63]. Convergence of the Gibbs sampler can be thought of as the Markov chain reaching the stationary distribution.

5.3.2 Variational Inference

In the previous section we discussed Gibbs sampling [17] which is a Markov Chain Monte Carlo (MCMC) method. MCMC methods form part of the stochastic route to finding a solution for the posterior distribution of the latent variables. Variation inference (VI) or variational Bayes falls into the deterministic solution of inference. The basic idea is to pick an approximation $q(z;v)$ with variational parameter v from some tractable family, and then try to make this approximation as close as possible to the true posterior distribution $p(z|x)$, usually by minimising the Kullback-Leibler divergence $KL(q(z;v)||p(z|x))$ from the posterior to the approximate distribution [37]. This is illustrated in Figure 5.3.

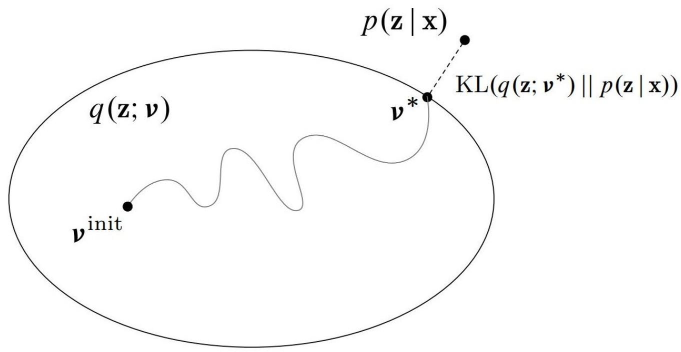


Figure 5.3: Variational inference illustration

5.4 Evaluation Methods

The methods described in this Chapter require metrics to measure their performance. However there is no way to measure performance using model parameters for topic models on a task . This is a result of having no ground truth for unstructured text [12]. For specific applications such as IR and document classification there exist evaluation measures [12, 58], these are extrinsic evaluation measures. There are also intrinsic measures of performance, these metrics are independent of any application and measure the quality of the models based on held-out data previously unseen to the model. When using perplexity as a intrinsic evaluation measure for topic models we must think of the topic models as language models and they are bad language models due to the bag-of words assumption [37]. In [58] there are other held-out probability estimation methods that are discussed that can be explored, but are outside of the scope of this study. Held-out probability methods for evaluating topic models have been criticised [10] and a coherence measure was proposed to measure human interpretability of topics models automatically. There are also human evaluation methods that have been described in evaluating topics such as word intrusions and topic intrusions [10]. For our applications we rely on extrinsic measures since IR is the application considered for this study. We still discuss perplexity in the next section.

5.4.1 Perplexity

The perplexity of a language model on a held-out dataset is the inverse probability of the held-out dataset, normalised by the number of words [24]. For a test set $d_t = \{w_{ti}\}_{i=1}^{N_{d_t}}$:

$$\begin{aligned} Perplexity(d_t) &= P(w_{t1}, w_{t2}, \ldots, w_{tN_{d_t}})^{-\frac{1}{N_{d_t}}} \\ &= \sqrt[N_{d_t}]{\tfrac{1}{P(w_{t1}, w_{t2}, \ldots, w_{tN_{d_t}})}} \end{aligned} \tag{5.1}$$

Using the chain rule we get:

$$Perplexity(d_t) = \sqrt[N_{d_t}]{\prod_{i}^{N_{d_t}} \frac{1}{P(w_{ti}|w_{t1}, \ldots, w_{ti-1})}} \tag{5.2}$$

To measure the performance of the models we want to assign a higher probability to the test set. This means that the model is accurately predicting the test set. Note that a higher probability means a smaller perplexity value. Therefore maximising the fit of the model to the test set is the same as minimising the perplexity of the model [24]. For evaluating topic models, perplexity is useful for model selection and can measure the relative performance between topic models as the number of chosen topics [12]. However, heavy criticism has been place on perplexity as an intrinsic evaluation measure for topic models [10, 12]. This is in part due to the perplexity's dependence on the vocabulary size being modelled. This means we cannot use it to compare models with different input features or that have different input languages. We then investigate coherence and how it differs from the measure we have just discussed.

5.4.2 Coherence

Though perplexity has is useful as a tool to pick parameters for a topic model or to choose which models to use for a collection. The held-out log-likelihood is not a good representative for the quality of topic model produced by the model [35,38,47]. The goal is to develop a measure that will reflect the interpretability of the topic produced by a topic models when presented to a human. This then leads to the overall consumption of topic models by the end user. We discuss the different measures of coherence below which are reviewed in [47]. We start with a measure introduced in [38] :

$$\begin{aligned} C_{UCI} &= \tfrac{2}{N(N-1)} \textstyle\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} PMI(w_i, w_j) \\ PMI(w_i, w_j) &= \log \tfrac{P(w_i, w_j)+\epsilon}{P(w_i) \cdot P(w_j)} \end{aligned} \tag{5.3}$$

The measure is based on point-wise mutual information (PMI) and in the results presented in [38], this measure correlated most with human judgement of coherent topics. The probabilities are based on word co-occurrence counts. Another variant of coherence measure was discussed in [35] which accounted for the ordering of the words in each topic. We measure this coherence in the following way :

$$C_{UMass} = \frac{2}{N(N-1)} \sum_{i=2}^{N-1} \sum_{j=1}^{i-1} log \frac{P(w_i, w_j) + \epsilon}{P(w_j)} \tag{5.4}$$

We limit our discussion to these two measures of coherence, we refer to the reader to [47] for further reading. We have discussed evaluation methods for topic modelling and move on to the estimation of the latent variable.

5.4.3 Conclusion

In this section we discussed the latent variable models namely; probabilistic latent semantic index and latent Dirichlet allocation, which are both generative models. Where the LDA is a Bayesian graphical model in that it places priors on the hyper parameters. We also describe evaluation methods for the models in the way of perplexity and coherence. We also touch on the estimation of parameters using MCMC methods and variational inference. We move on to discuss how these measures can be used in the application of relevance judgement and information retrieval.

Chapter 6

Application

In this chapter we apply both models on datasets in order to assess their ability to produce appropriate latent semantic representations of corpora. We use two datasets, namely, the popular baseline 20 newsgroup[1] data and a fake news data sets (https://www.kaggle.com/mrisdal/fake-news). Although our methods are unsupervised, the labels associated with documents in the datasets provide a ground truth of semantically relevant and irrelevant documents. We can therefore assess whether a model is able to identify relevant documents from irrelevant ones for previously unseen data in an unsupervised fashion.

6.1 Datasets

6.1.1 20 Newsgroup

We use two datasets these are the 20-newsgroup and the fake news data sets. The 20-newsgroup data[2] contains 18846 newsgroup documents split between 20 categories. This data set is broken up into two, the training set for parameter estimation and development and the test set for model performance evaluation. We access this data set using the sci-kit learn library in Python which is equipped with functions that make data extraction and loading simple. In Figure 6.1 we find the distribution of documents in for each of the categories in the training set.

[1]http://qwone.com/ jason/20Newsgroups/
[2]http://archive.ics.uci.edu/ml/datasets/twenty+newsgroups

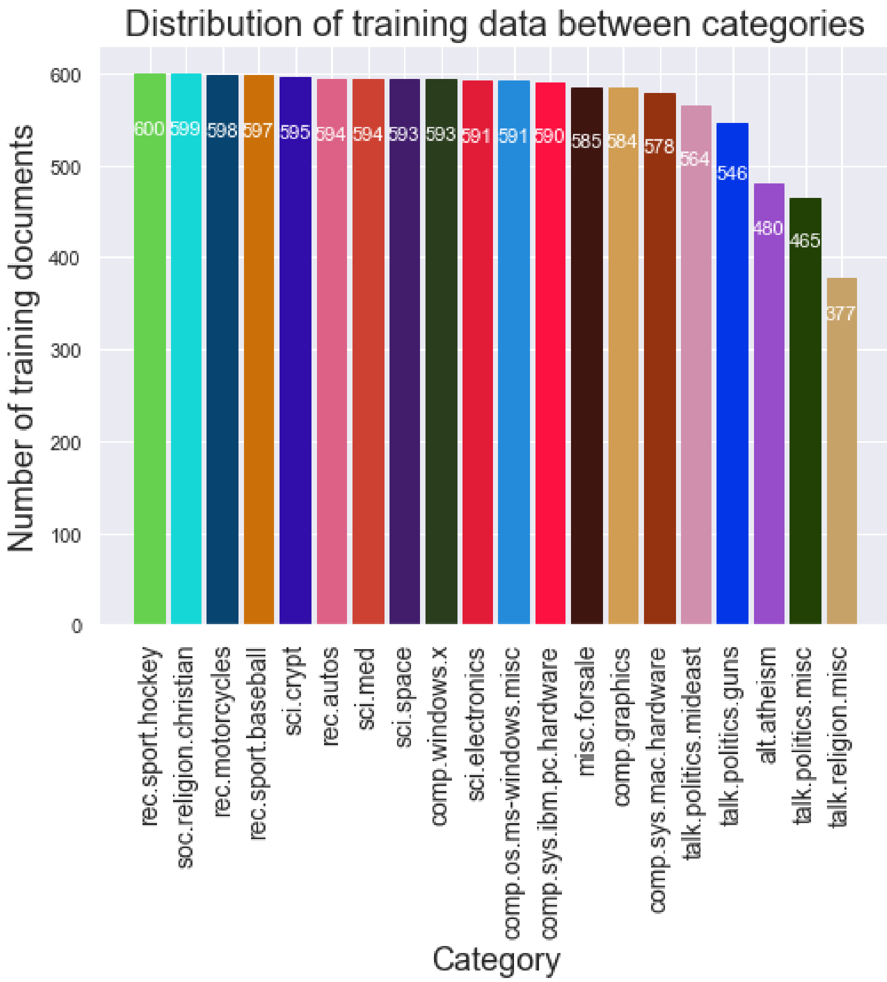


Figure 6.1: Category distribution for training data

6.1.2 Fake News

The dataset contains text and metadata from 244 pages, comprising a total of 12,999 comments. The data is collected using the webhose.io API. Each website was labeled according to the BS Detector. A 'bs' tag was applied to data sources that lacked a label. No real, credible or trustworthy news sources are identified in this dataset. This BS Detector is a Google Chrome Extension made by Daniel Sieradski (https://www.kaggle.com/mrisdal/fake-news).

6.2 Data Preparation

The prepossessing phase of modelling converts the original textual data to a machine readable format, where the most significant text features that are selected to differentiate between categories are identified [54]. Text data are very noisy and the preprocessing stage is crucial in order to reduce noise and improve the quality of the model. The colloquial phrase “Garbage in, garbage out” gives meaning to why we must first clean our data before any modelling is done. The following steps constitute the basic functions in data preparation for text:

1. Text normalisation
2. Removal of stopwords
3. Tokenisation
4. Stemming / Lemmatisation

6.2.1 Text normalisation

The process of text normalisation aim to cleans an input word or sentence by transforming all non-standard lexical or syntactic variation into their standard dictionary form [36]. This phase of data preparation includes but is not limited to:

- Converting all letters to lower or uppercase
- Converting numbers into words or removing them
- Removing punctuation, accent marks and other diacritics[3]
- Removing leading and trailing spaces

6.2.2 Removal of stopwords

Stopwords are frequent words that have been proven to carry no information. In the context on language specific stopword we refer to functional words such as pronouns, propositions and conjugations. The impact of stopwords in text processing is mainly related to term weighting [14]. This effect to term weighting is from the frequency difference of stopwords to other words in the corpus. Their frequency can also cause problems in efficiently processing text since they contribute little information. The removal of stopwords can increase the efficiency of the indexing process since they form 30 to 50% of tokens in large text [14]. 6.2.

[3] A sign, such as an accent or cedilla, which when written above or below a letter indicates a difference in pronunciation from the same letter when unmarked or differently marked. Found in languages such as Setswana and Tshivenda in the South African context

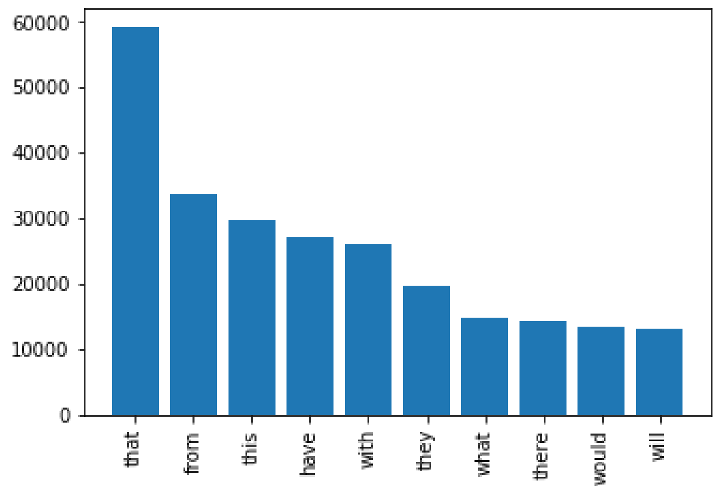


Figure 6.2: Stopword distribution for 20newsgroup

Other areas of contempt for modelling text are short and empty documents. We find how many documents are empty in our training set. We find that there are 218 documents that contain zero characters. This is 1.93% of our training data. For documents shorter than 150 words we find there are 8027 documents from the 11314 documents in our training set. We observe the distribution of short words in figure 6.3.

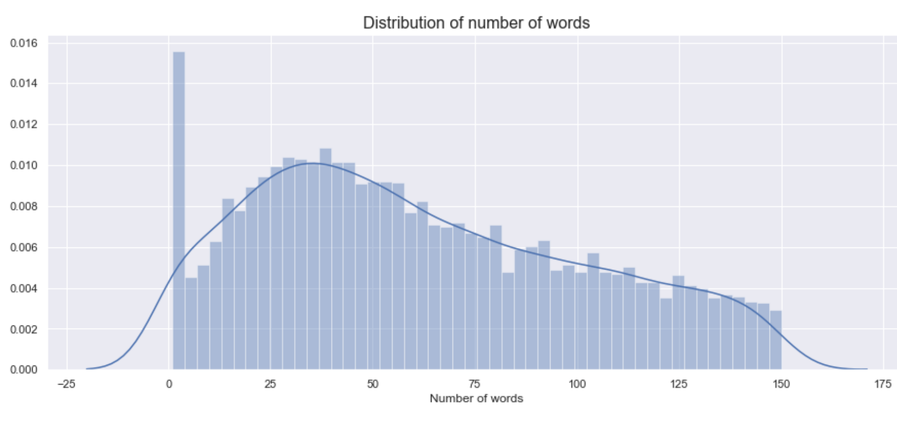


Figure 6.3: Distribution of short text in training set

For our study we consider documents that have fewer than 30 words as short documents and remove the 218 empty documents.

6.2.3 Tokenisation

Tokensation is the process of breaking up a sequence of characters in text by locating word boundaries, the point where one word ends and another begins [41]. The result is broken up strings, called tokens. This step of data preparation allows the document to be broken down to units that constitute it. With the removal of stopwords and text normalisation the result of tokenisation is a bag of words. From this bag we can construct a vector representation of the document by using the frequency as a weight for each index term in the document. The issues of tokenisation are language specific. Thus there are approaches to tokenisation for space delimited languages and approached for unsegmented languages [41]. European languages are space delimited languages in which a space insertion indicates a word boundary. While Japanese, Chinese and Thai are unsegmented languages and there is a succession of words without spaces between them. When tokenisation is more challenging and difficult to capture in a few rules, a machine learning approach can be useful. In this case tokenisation is treated as a character classification problem or a sequential labelling problem[4].

6.2.4 Stemming and Lemmatisation

A stemming algorithm is a computational procedure which reduces all words of the same root to a common form, usually by stripping each word of its derivational and inflectional suffixes [29]. While lemmatization refers to the use of vocabulary and morphological analysis of words to try and remove inflectional endings and return words to their dictionary form [3]. There are many stemming algorithms, but in this study Porter's stemmer is considered since its simple approach to conflation (mapping similar stems together) seem to work well in practice and it is applicable to a range of languages [62]. Applying a stemming algorithm in data preparation ensures that there is a reduction in the number of words being indexed [3], thus reducing our feature space and this may have an impact on retrieval performance. Lemmatization on the other hand, analyzes whether words in the query are nouns or verbs. It also increases the retrieval of relevant documents through the use of synonyms. Lemmatization also reduces the feature space and has shown to improve retrieval performance as result [3]. Further a comparison between stemming and lemmatization reveals that lemmatization outperforms stemming [3]. This may be a result of limitations inherent in the stemming algorithm, in that it has no access to information about grammatical and semantic relation for each word being processed [29]. Where as lemmatization is more advanced since it considers morphological analysis and has access to word synonyms unlike stemming [3].
The application of the aforementioned data preparation methods result in a sparse *document* $\times$ *word* matrix. An extra step of filtering words based on minimum and maximum frequency is applied. This step will result in less sparse

[4] https://uclmr.github.io/stat-nlp-book-scala/01_tasks/00_tokenization.html

matrix which results in better model performance [30]. The data preparation coding is done in the NLP package gensim [46] in Python 3.7.

6.3 Document Similarity

We train the LDA and word2vec models on 20 newsgroup data to observe how each perform in measuring similarity between categories. We use soft cosine similarity to measure the distance between vectors. We train both models on the entire training set from different categories and hold out some documents for testing. We use the gensim library from python to train both models, and we compare the distance distributions for each models between categories. We also look at the Kolmogorov-Smirnov test to see whether the two distance distributions are significantly different or not. Using some of the preprocessing techniques that have been discussed in Section 6.2, we clean the documents and filter out for short documents that contain 20 or less words. Figure 6.4 contains the lengths of the documents in the collection.

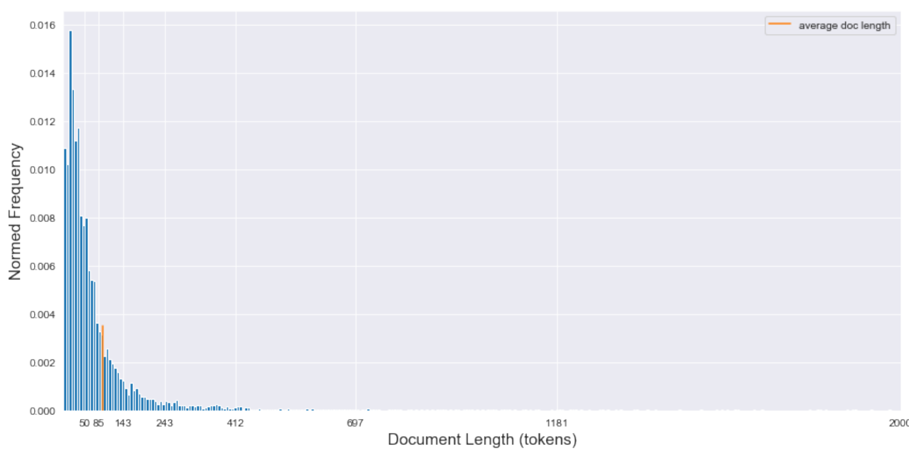


Figure 6.4: Distribution for documents lengths

6.3.1 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov Test or K-S test is a non parametric test that measures the the maximum distance between distributions. The Kolmogorov- Smirnov statistic quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution, or between the empirical distribution functions of two sample[5]. The result is a statistical approach to determine whether two distributions are generated from samples coming from the same population. This test makes no assumptions of

[5]https://towardsdatascience.com/Kolmogorov-Smirnov-test-84c92fb4158d

normality for the distributions. We formally define this test. Given the cumulative distribution $F_0(x)$ of the hypothesized distribution and the empirical distribution $F_{data}(x)$ of the observed data, the Kolmogorov- Smirnov test is given by,

$$D = \sup_x |F_0(x) - F_{data}(x)|$$

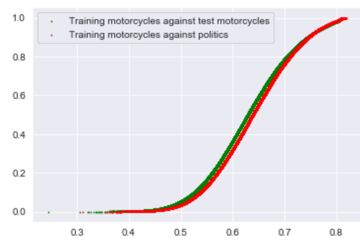
Discussion on the differences on the distribution of this statistic when we are working with continuous versus when we are working with discrete distributions [1] which is beyond the scope of this study.

6.4 Experimental Setup

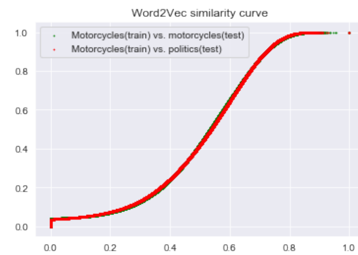
We train the LDA model with paratmeters α =0.001 , β = 0.005 and K =100. We use the Gensim [45] library in Python 3.7. We choose a reference category to form a ground truth for our similarity measures. For the LDA we use the Jensen-Shannon divergence to evaluate similarity. We compare the results of the LDA to word2vec and use the soft-cosine similarity measure. The word2vec is also found in the gensim library with a window size of 5 and a variable dimension size for the word vectors based on the number of documents in the our training set. We use the continuous bag-of-words [33] or CBOW algorithm for training the neural network. We then investigate the relevance judgement of each of the model by measuring the minimum score assigned to the reference corpus and how many non relevant documents were discarded as a result.

Experiment 1: Investigate average similarity distance measures for the LDA and word2vec to evaluate semantic relatedness :

We investigate similarity performance of the LDA and compare it to that of the word2vec with the motorcycle category as the reference topic and the auto and politics categories as tests. We expect the similarity between the auto category and the reference to be higher than that of the reference and the politics category since the latter are compose of completely different vocabularies. We inspect the average Jensen-Shannon divergence and compare these results to the mean soft-cosine distances for the word2vec. We inspect the Kolmogorov- Smirnov test results to see whether the results from Figure 6.5a, Figure 6.5b, Figure 6.6a and Figure 6.6b are statistically significant. We implement ks-2samp function from the stats package in the Scipy library in Python 3.7 for the results in Table 6.1.



(a) Motorcycle vs. politics for LDA



(b) Motorcycle vs. politics for word2vec

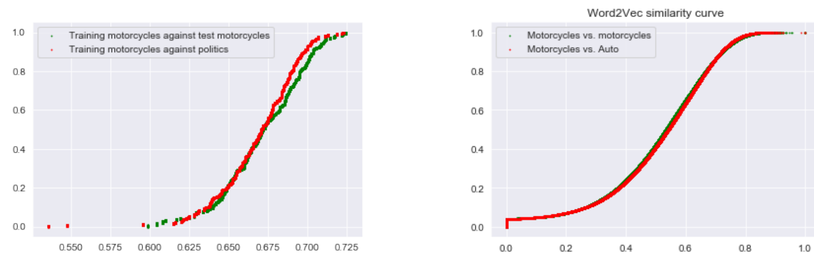
Figure 6.5: Average distance distribution

Model	K-S statistic	p-value
LDA(Motorsport v Politics)	04720	6.9400e-68
W2V(Motorsport v Politics)	4.4962e-06	1.0
LDA(Motorsport v Auto)	0.1050	1.2817e-306
W2V(Motorsport v Auto)	4.4720e-06	1.0
LDA(Tech v Religion)	0.3163	0
W2V(Tech v Religion)	6.9283e-06	1

Table 6.1: Kolmogorov-Smirnov Test (Motorsport v. Auto v. Politics)

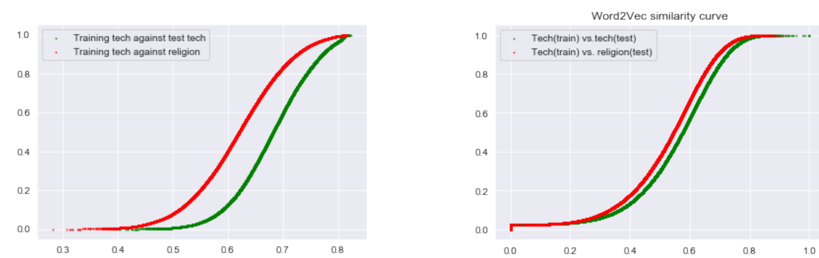
Looking at the K-S statistic and the p-value for both cases of the LDA, we reject the null hypothesis that the average similarity between motorsport and politic is identical. And the average similarity of the auto category is identical to the motorsport. We can suggest that this may be true since both categories are not actually identical to the reference though the auto labelled data are similar. For the word2vec we inspect both the K-S statistic and the p-value and in both cases we not reject the null hypothesis.

It seems for word embeddings all the categories are represented with similar word vectors and the soft-cosine distance scores documents from the motorsport identically to those that come from the politics and auto category.



(a) Motorcycle vs. Auto for LDA (b) Motorcycle vs. auto for word2vec

Figure 6.6: Distance distribution for similar categories



(a) Technology vs.religion for LDA (b) Technology vs.religion for word2vec

Figure 6.7: Distance distribution for different categories

We observe that if we change the reference category to the technology category we have Figure 6.7a for the LDA and Figure 6.7b for word2vec. We see that the LDA in this regard has shorter average distances for the religion category than it has for the reference category documents. We observe that for the word2vec these distances are much closer than that of the LDA, but we have that the religion category still has shorter average similarity distances when compared to the reference category. We move on to measures of evaluation to tell how well our models are performing, using more intrinsic evaluation methods.

Experiment 2: Investigating whether semantic similarity is connected to retrieval of relevant documents

We take an application of relevance judgement or retrieval as the extrinsic method of evaluation and connect it to the average distances we inspected in Experiment 1. We aim to be able to form a base line with this analysis of average distance distributions. For this we look at the fake news data. We start with Figure 6.8a and Figure 6.13b. These are the average distance pseudo CDFs for the LDA and word2vec. We observe two different trends, with the reference category in green(Reliable news sources) and the unreliable news having longer average distances when compared to the reference.

In Figure 6.8a we observe that the LDA have shorter average distances for the reliable news as compared to the unreliable news. The same is true of the word2vec in Figure 6.13b. We the see if this affects the models ability to pick those documents that are relevant from the rest of the corpus.

(a) (b)

Figure 6.8: Distance distribution between models

In Figure 6.9 we have the framework which will inform us of how each of the models tested on a data set perform

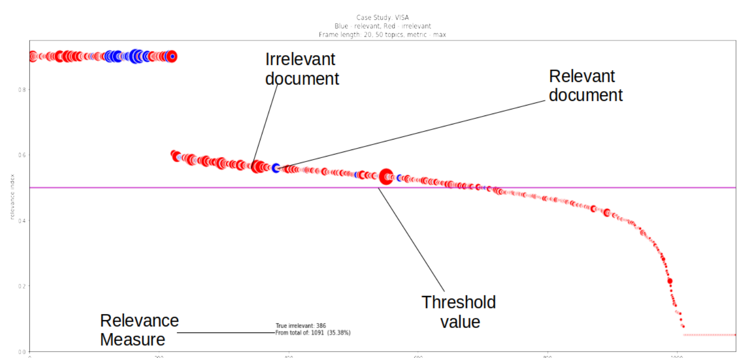


Figure 6.9: Relevance plot for fake news data LDA

For Figure 6.10 we have that though this model has a steeper average distance for documents in the reliable class, the minimum score given for the true relevant documents only allows for 30 true irrelevant documents to be discarded from 1919. This is equivalent to 1.56% documents automatically discarded based on the score allocated by the LDA. The word2vec does a better job.

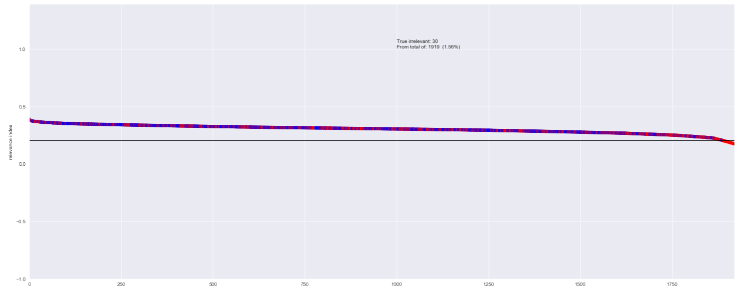


Figure 6.10: Relevance plot for fake news data LDA

As can be seen in Figure 6.11 the word2vec does a better job separating relevant documents from that are irrelevant. It manages to discard 60 true irrelevant documents from 2050 documents. This means 2.98% of irrelevant documents are discarded automatically based on the score given by the word2vec.

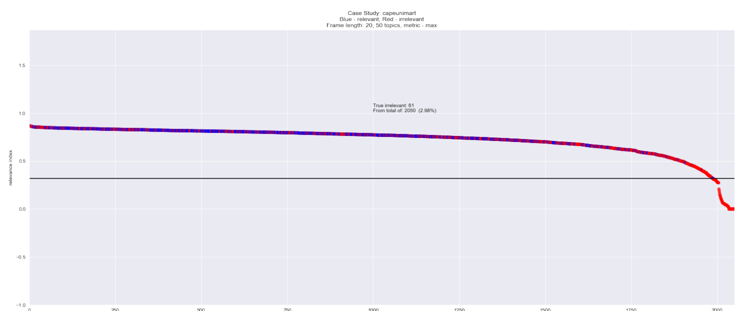
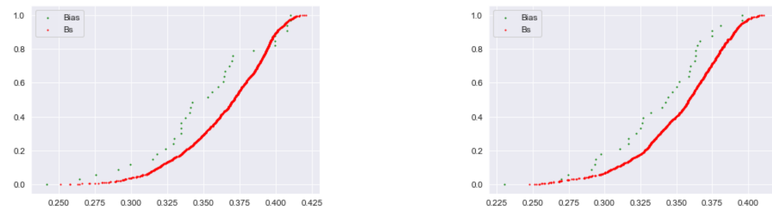


Figure 6.11: Relevance plot for fake news data word2vec

Experiment 3: Investigating the effect K on semantic similarity and to retrieval of relevant documents:

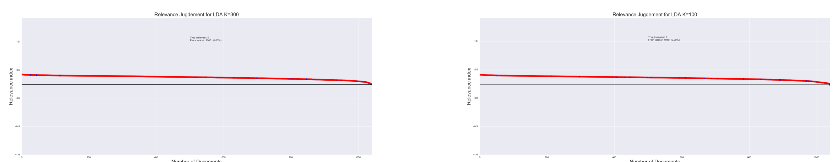
We observe the the effect of the number of topics chosen for the LDA model on the fake news data set, we have 12999 documents and we remove any documents that have less that 30 tokens and we remain with 11340 documents that we split 90/10 into the training and test set. We set a seed value to keep consistent results and keep all the corpus parameters α and β fixed. We then get the average Jensen-Shannon divergence minus 1 as the similarity measure and observe the difference in distance between the 'Bias' category and the 'BS' category in the data set.



(a) (b)

Figure 6.12: Distance distribution for LDA

There are 34 Bias documents in the entire corpus, significantly less to those of the BS documents. But for both K values we have that the Bias distances are relatively smaller than those of the BS when compared to the training corpus. We observe whether this has any effect on the retrieval of relevant documents in our corpus using the 1157 test documents. We observe the following results:



(a) (b)

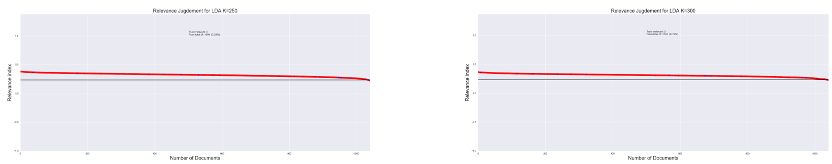
Figure 6.13: Distance distribution for LDA

For both models we have a zero percent dicarding of true irrelevant or in this case BS documents. We look at the more interesting case of $K = 250$ and $K = 300$ in the following:

(a) (b)

Figure 6.14: Distance distribution for LDA

In this case we observe the BS and Bias categories to be much closer than in 6.12. But let us observe what this means in terms of retrieval in the following:



(a) (b)

Figure 6.15: Distance distribution for LDA

We observe that for $K = 250$ the LDA has a better performance than when we have $K = 300$, but we cannot observe from 6.14 a distinguishable way to deduce the results in 6.15. We move on to conclude the results presented in this work and would be possible next steps.

Chapter 7

Conclusion

7.1 Evaluation and outcomes

We investigate the performance of LDA - a probabilistic representation and word2vec - a prediction-based representation on the task of semantic representation. The objective of these experiments is to evaluate the respective model's ability to capture the latent semantic space of a corpus. We implement the LDA and word2vec models on 20newsgroups and fake news data. In experiment 1, we visualise the cumulative semantic distances. It seems from the final set of experiments (represented in Figure 6.7a) that LDA produces shorter distances between the training and test sets for the same category than between different categories. Both datasets are labelled, which enables us to use a ground truth: The observations (documents in our case) are labelled as relevant (blue) and not relevant (red). We introduce a relevance graph based on a minimum threshold value for what we label as the true relevant data. The threshold value acts as a decision boundary in order to determine the predicted labels of the test documents. Our results prove that the word2vec model have a better classification rate than the LDA.

7.2 Contribution

High dimensional data such as text are often labelled as big data, not only because of high volumes, but also because of veracity and velocity. It is for these reasons that unsupervised representations are becoming more in demand in order to project the data onto a lower dimensional space that is more manageable. In this work, we packaged a well known topic model, LDA as a distributional semantic model. We compare LDA's ability to model a corpus' semantics *and* to distinguish between corpora to a a well established model - word2vec. LDA shows promise in the task of dimensionality reduction and semantic representation of a corpus. It is clear from experiments that LDA performs better in scenarios where the categories are decisively different as oppose to where the differences are more subtle.

7.3 Future Work

We recommend the following considerations be made on any future work based on this study:

- Even though perplexity is a point of contention in the topic modelling community as an intrinsic measure, it maybe of value to explore it as an intrinsic evaluation measure for both models
- The use of the pLSI in search engine optimisation may suggest it being suitable in a comparative study to observe whether the result provided by the LDA model can be relied on.
- Parameter tunning for both models: By this we mean using the best estimates for both models on the training data that is available to the model. This can provide more insight as to which model performs the best.
- Exploring the effect of the chosen number of topics on the performance when compared to the word2vec.
- Lastly, exploring models such as the Hierarchical Latent Dirichlet model on the same task could be of benefit since this remove the influence on modelling as a result if the number of topics chosen by the practitioner.

7.4 Conclusion

In the task of retrieval we seek automated solutions to cut down on costs and improve efficiency. This can be achieved by the use of machine learning algorithms. With this work we explored a generative, probabilistic approach to achieve this objective. We illustrated that LDA achieve satisfactory results in this task when semantics between a relevant corpus and irrelevant query are significantly different. The results encourages further investigation into this topic.

Bibliography

[1] Taylor B Arnold and John W Emerson. Nonparametric goodness-of-fit tests for discrete null distributions. *R Journal*, 3(2), 2011.

[2] Sanjeev Arora, Rong Ge, Yoni Halpern, David M Mimno, Ankur Moitra, David Sontag, Yichen Wu, and Michael Zhu. Learning topic models-provably and efficiently. *Commun. ACM*, 61(4):85–93, 2018.

[3] Vimala Balakrishnan and Ethel Lloyd-Yemoh. Stemming and lemmatization: a comparison of retrieval performances. 2014.

[4] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 238–247, 2014.

[5] T. Beysolow. *Applied Natural Language Processing with Python: Implementing Machine Learning and Deep Learning Algorithms for Natural Language Processing*. Apress, 2018.

[6] Taweh Beysolow II. Applied natural language processing with python.

[7] R. Bilbro, T. Ojeda, and B. Bengfort. *Applied Text Analysis with Python*. O'Reilly Media, Incorporated, 2018.

[8] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[9] David Meir Blei. *Probabilistic models of text and images*. PhD thesis, Citeseer, 2004.

[10] Jonathan Chang, Sean Gerrish, Chong Wang, Jordan L Boyd-Graber, and David M Blei. Reading tea leaves: How humans interpret topic models. In *Advances in neural information processing systems*, pages 288–296, 2009.

[11] Jackie Chi Kit Cheung and Gerald Penn. Probabilistic domain modelling with contextualized distributional semantic vectors. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 392–401, 2013.

[12] Aletta Johanna De Waal. *Topic Models with Structured Features*. PhD thesis, North-West University, Potchefstroom Campus, 2010.

[13] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[14] Ibrahim Abu El-Khair. Effects of stop words elimination for arabic information retrieval: a comparative study. *International Journal of Computing & Information Sciences*, 4(3):119–133, 2006.

[15] Christos Faloutsos and Douglas W Oard. A survey of information retrieval and filtering methods. Technical report, 1998.

[16] Hagen Fürstenau and Mirella Lapata. Semi-supervised semantic role labeling. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 220–228. Association for Computational Linguistics, 2009.

[17] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, (6):721–741, 1984.

[18] Thomas L Griffiths and Mark Steyvers. A probabilistic approach to semantic representation. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 24, 2002.

[19] Thomas Hofmann. Probabilistic latent semantic indexing. In *ACM SIGIR Forum*, volume 51, pages 211–218. ACM, 2017.

[20] Keith J Holyoak. Parallel distributed processing: explorations in the microstructure of cognition. *Science*, 236:992–997, 1987.

[21] Anna Huang. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, volume 4, pages 9–56, 2008.

[22] Jay J Jiang and David W Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *arXiv preprint cmp-lg/9709008*, 1997.

[23] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[24] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[25] Thomas K Landauer, Peter W Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.

[26] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[27] Baoli Li and Liping Han. Distance weighted cosine similarity measure for text classification. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 611–618. Springer, 2013.

[28] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

[29] Julie Beth Lovins. Development of a stemming algorithm. *Mech. Translat. & Comp. Linguistics*, 11(1-2):22–31, 1968.

[30] Jocelyn Mazarura, Alta de Waal, Frans Kanfer, and Sollie Millard. Topic modelling for short text. *Proceedings of the Pattern Recognision Association of South Africa (PRASA 2014), 2014*, 2015.

[31] Rada Mihalcea, Courtney Corley, Carlo Strapparava, et al. Corpus-based and knowledge-based measures of text semantic similarity. In *Aaai*, volume 6, pages 775–780, 2006.

[32] Kenta Mikawa, Takashi Ishida, and Masayuki Goto. A proposal of extended cosine measure for distance metric learning in text classification. In *2011 IEEE International Conference on Systems, Man, and Cybernetics*, pages 1741–1746. IEEE, 2011.

[33] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[35] David Mimno, Hanna M Wallach, Edmund Talley, Miriam Leenders, and Andrew McCallum. Optimizing semantic coherence in topic models. In *Proceedings of the conference on empirical methods in natural language processing*, pages 262–272. Association for Computational Linguistics, 2011.

[36] Alejandro Mosquera, Elena Lloret, and Paloma Moreda. Towards facilitating the accessibility of web 2.0 texts through text normalisation. In *Proceedings of the LREC workshop: Natural Language Processing for Improving Textual Accessibility (NLP4ITA)*, pages 9–14, 2012.

[37] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[38] David Newman, Jey Han Lau, Karl Grieser, and Timothy Baldwin. Automatic evaluation of topic coherence. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 100–108. Association for Computational Linguistics, 2010.

[39] Kamal Nigam, Andrew Kachites McCallum, Sebastian Thrun, and Tom Mitchell. Text classification from labeled and unlabeled documents using em. *Machine learning*, 39(2-3):103–134, 2000.

[40] Vít Novotnỳ. Implementation notes for the soft cosine measure. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 1639–1642. ACM, 2018.

[41] David D Palmer. Tokenisation and sentence segmentation. *Handbook of natural language processing*, pages 11–35, 2000.

[42] Siddharth Patwardhan, Satanjeev Banerjee, and Ted Pedersen. Using measures of semantic relatedness for word sense disambiguation. In *International conference on intelligent text processing and computational linguistics*, pages 241–257. Springer, 2003.

[43] Jefferson Provost. Naıve-bayes vs. rule-learning in classification of email. *University of Texas at Austin*, 1999.

[44] GN Ramadevi and K Usharani. Study on dimensionality reduction techniques and applications. *Publications of Problems & Application in Engineering Research–Paper*, 4, 2013.

[45] Radim Rehurek and Petr Sojka. Software framework for topic modelling with large corpora. In *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Citeseer, 2010.

[46] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[47] Michael Röder, Andreas Both, and Alexander Hinneburg. Exploring the space of topic coherence measures. In *Proceedings of the eighth ACM international conference on Web search and data mining*, pages 399–408. ACM, 2015.

[48] Timothy N Rubin, America Chambers, Padhraic Smyth, and Mark Steyvers. Statistical topic models for multi-label document classification. *Machine learning*, 88(1-2):157–208, 2012.

[49] Gerard Salton. The smart environment for retrieval system evaluation–advantages and problem areas. 1981.

[50] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[51] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[52] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press, 2008.

[53] Grigori Sidorov, Alexander Gelbukh, Helena Gómez-Adorno, and David Pinto. Soft similarity and soft cosine measure: Similarity of features in vector space model. *Computación y Sistemas*, 18(3):491–504, 2014.

[54] V Srividhya and R Anitha. Evaluating preprocessing techniques in text categorization. *International journal of computer science and application*, 47(11):49–51, 2010.

[55] Joshua B. Tenenbaum. Bayesian modeling of human concept learning. In M. J. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 59–68. MIT Press, 1999.

[56] Benjamin Bengfort Tony Ojeda, Rebecca Bilbro. *Applied Text Analysis with Python*, chapter 4, page 1. O'Reilly Media, Inc., 2018.

[57] Patrick Peter TurneyPantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research 37*, 2010.

[58] Hanna M Wallach, Iain Murray, Ruslan Salakhutdinov, and David Mimno. Evaluation methods for topic models. In *Proceedings of the 26th annual international conference on machine learning*, pages 1105–1112. ACM, 2009.

[59] Chong Wang and David M Blei. Collaborative topic modeling for recommending scientific articles. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 448–456. ACM, 2011.

[60] Xuerui Wang, Andrew McCallum, and Xing Wei. Topical n-grams: Phrase and topic discovery, with an application to information retrieval. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 697–702. IEEE, 2007.

[61] Xing Wei and W Bruce Croft. Lda-based document models for ad-hoc retrieval. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 178–185. ACM, 2006.

[62] Peter Willett. The porter stemming algorithm: then and now. *Program*, 40(3):219–223, 2006.

[63] Scott L Zeger and M Rezaul Karim. Generalized linear models with random effects; a gibbs sampling approach. *Journal of the American statistical association*, 86(413):79–86, 1991.

.1 Code: Fake News Classifier Application

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from googletrans import Translator
from simplejson import JSONDecodeError
import re
from sklearn import metrics
import matplotlib.pyplot as plt
%matplotlib inline
#import the data
fake_news = pd.read_csv('/home/szmjali/Desktop/Research Code/Code for research/fake newstrain.csv')
fake_news_data=fake_news[['text','label']]
fake_news_data['text']=fake_news_data['text'].map(lambda x: re.sub('[^A-Za-z]+', ' ',str(x)))
fake_news_data=fake_news_data.dropna()
X_train,X_test,y_train,y_test = train_test_split(fake_news_data['text'],fake_news_data['label'],test_size = 0.2)
#from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
cvec = CountVectorizer(stop_words='english',min_df=10)
bag_of_words = cvec.fit_transform(X_train)
feature_names = cvec.get_feature_names()
bag_of_words_test=cvec.transform(X_test)
vectorized_text =pd.DataFrame(bag_of_words.A,
                 columns=cvec.get_feature_names())
vectorized_text_test =pd.DataFrame(bag_of_words_test.A,
                 columns=cvec.get_feature_names())

from sklearn.naive_bayes import MultinomialNB
nb = MultinomialNB()
nb_model = nb.fit(vectorized_text, y_train)
acc = nb_model.score(vectorized_text, y_train)
ratio_class1 = y_train.mean()
from sklearn import svm
clf=svm.LinearSVC()
svm_model=clf.fit(vectorized_text,y_train)
res= svm_model._predict_proba_lr(vectorized_text_test)
mnb_predicted = nb_model.predict(vectorized_text_test)
svm_predicted= svm_model.predict(vectorized_text_test)
mnb_probs = nb_model.predict_proba(vectorized_text_test)
mnb_acc_score = metrics.accuracy_score(y_test, mnb_predicted)
svm_acc_score = metrics.accuracy_score(y_test, svm_predicted)
```

```python
mnb_auc_score = metrics.roc_auc_score(y_test, mnb_probs[:, 1])
svm_auc_score = metrics.roc_auc_score(y_test, res[:, 1])
print(metrics.classification_report(y_test, mnb_predicted))
print(metrics.classification_report(y_test, svm_predicted))

fpr, tpr, thresholds = metrics.roc_curve(y_test, mnb_probs[:, 1], pos_label=1)
fpr2, tpr2, thresholds2 = metrics.roc_curve(y_test, res[:, 1], pos_label=1)

plt.figure()
lw = 2
plt.plot(fpr, tpr, color='red',
         lw=lw, label='MNB ROC curve (area = %0.2f)' % metrics.roc_auc_score(y_test, mnb_probs[:, 1]))
plt.plot(fpr2, tpr2, color='green',
         lw=lw, label='SVM ROC curve (area = %0.2f)' % metrics.roc_auc_score(y_test, res[:, 1]))

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve of SVM v. MNB')
plt.legend(loc="lower right")
plt.show()

def plot_confusion_matrix(y_true, y_pred, classes,
                          normalize=False,
                          title=None,
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = metrics.confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)
```

```python

    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           # ... and label them with the respective list entries
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax

mnb_cm = metrics.confusion_matrix(y_test, mnb_predicted)
svm_cm = metrics.confusion_matrix(y_test, svm_predicted)

plot_confusion_matrix(y_test, mnb_predicted, classes=[0,1],
                      title='Confusion matrix(MNB), without normalization')

plot_confusion_matrix(y_test, svm_predicted, classes=[0,1],
                      title='Confusion matrix(SVM), without normalization')
```

Code: 20Newsgroups multi-class classification Application

```python
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer,CountVectorizer
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics
```

```
from sklearn import svm
newsgroups_train = fetch_20newsgroups(subset='train')
categories = ['rec.motorcycles', 'talk.religion.misc','talk.politics.mideast','rec.autos','comp.sys.ibm.pc.hardwa
newsgroups_train = fetch_20newsgroups(subset='train',categories=categories)
newsgroups_test = fetch_20newsgroups(subset='test',categories=categories)
cvec = CountVectorizer(stop_words='english',min_df=10)
bag_of_words=cvec.fit_transform(newsgroups_train.data)
vectorizer = TfidfVectorizer(max_df=0.5,min_df=2,
                                   ngram_range=(1,2),
                                   stop_words='english',
                                   token_pattern=r'\b[^\d\W]+\b')

vectors = vectorizer.fit_transform(newsgroups_train.data)
bag_of_words_test=cvec.transform(newsgroups_test.data)
vectors_test = vectorizer.transform(newsgroups_test.data)
nbm_clf = MultinomialNB(alpha=.01)
y_score=nbm_clf.fit(vectors, newsgroups_train.target)
nbm_pred = nbm_clf.predict(vectors_test)
probs = nbm_clf.predict_proba(vectors_test)
clf=svm.LinearSVC()
svm_model=clf.fit(vectors,newsgroups_train.target)
svm_pred = clf.predict(vectors_test)
print(metrics.classification_report(newsgroups_test.target, nbm_pred, target_names=categories))
print(metrics.classification_report(newsgroups_test.target, svm_pred))
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_curve, auc
from sklearn.multiclass import OneVsRestClassifier
plt.figure()
lw = 2
n_classes=len(categories)
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = metrics.roc_curve(newsgroups_test.target, probs[:,i],pos_label=1)
    roc_auc[i] = metrics.auc(fpr[i], tpr[i])
colors = ['blue', 'red', 'green','yellow','purple','navy']
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=lw,
             label='{0} (area = {1:0.2f})'
             ''.format(categories[i], roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=lw)
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
```

```python
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for multi-class data')
plt.legend(loc="lower right")
plt.show()
def plot_confusion_matrix(y_true, y_pred, classes,
                          normalize=False,
                          title=None,
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = metrics.confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    #print(cm)

    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           # ... and label them with the respective list entries
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
```

```
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax
    import numpy as np
from sklearn.utils.multiclass import unique_labels
plot_confusion_matrix(newsgroups_test.target, nbm_pred, classes=[0,1,2,3,4],
                      title='Confusion matrix, without normalization for Multinomial naive Bayes'
plot_confusion_matrix(newsgroups_test.target, svm_pred, classes=[0,1,2,3,4],
                      title='Confusion matrix, without normalization for Support Vector Machine')
```

Fake News LDA Implementation

```
#import the required libraries for the experiment
import pandas as pd
import numpy as np
import nltk
from nltk.corpus import stopwords
import gensim
from gensim.models import LdaModel
from gensim import models, corpora, similarities
import re
from nltk.stem.porter import PorterStemmer
import time
from nltk import FreqDist
from scipy.stats import entropy
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set_style("darkgrid")
from sklearn.datasets import fetch_20newsgroups
from nltk.stem import WordNetLemmatize
df1 = pd.read_csv('fake.csv')
fake_data=df1[['text','type']]
def initial_clean(text):
    """
    Function to clean text of websites, email addresess and any punctuation
    We also lower case the text
    """
    text = re.sub('\s+', ' ', str(text))
    text = re.sub("((\S+)?(http(s)?)(\S+))|((\S+)?(www)(\S+))|((\S+)?(\@)(\S+)?)", " ", text)
    text = re.sub("[^a-zA-Z ]", "", text)
```

```python
    text = text.lower() # lower case the text
    text = nltk.word_tokenize(text)
    return text

stop_words = stopwords.words('english')
def remove_stop_words(text):
    """
    Function that removes all stopwords from text
    """
    return [word for word in text if word not in stop_words]

lmtzr = WordNetLemmatizer()
def stem_words(text):
    """
    Function to stem words, so plural and singular are treated the same
    """

    text = [lmtzr.lemmatize(word) for word in text]
    text = [word for word in text if len(word) > 1] # make sure we have no 1 letter words
    return text

def apply_all(text):
    """
    This function applies all the functions above into one
    """
    return stem_words(remove_stop_words(initial_clean(text)))

def jensen_shannon(query, matrix):
    """
    This function implements a Jensen-Shannon similarity
    between the input query (an LDA topic distribution for a document)
    and the entire corpus of topic distributions.
    It returns an array of length M where M is the number of documents in the corpus
    """
    # lets keep with the p,q notation above
    p = query[None,:].T # take transpose
    q = matrix.T # transpose matrix
    m = 0.5*(p + q)
    return np.sqrt(0.5*(entropy(p,m) + entropy(q,m)))

def get_most_similar_documents(query,matrix,k=10):
    """
    This function implements the Jensen-Shannon distance above
    and retruns the top k indices of the smallest jensen shannon distances
    """
    sims = jensen_shannon(query,matrix) # list of jensen shannon distances
    return sims.argsort()[:k] # the top k positional index of the smallest Jensen Shannon distances
```

```
def keep_top_k_words(text):
    return [word for word in text if word in top_k_words]

def train_lda(data):
    """
    This function trains the lda model
    We setup parameters like number of topics, the chunksize to use in Hoffman method
    We also do 2 passes of the data since this is a small dataset, so we want the distributions to stabilize
    """
    num_topics = 100
    chunksize = 300
    dictionary = corpora.Dictionary(data['tokenized'])
    corpus = [dictionary.doc2bow(doc) for doc in data['tokenized']]
    t1 = time.time()
    # low alpha means each document is only represented by a small number of topics, and vice versa
    # low eta means each topic is only represented by a small number of words, and vice versa
    lda = LdaModel(corpus=corpus, num_topics=num_topics, id2word=dictionary,
                   alpha=1e-2, eta=0.5e-2, chunksize=chunksize, minimum_probability=0.0, passes=2)
    t2 = time.time()
    print("Time to train LDA model on ", len(df), "articles: ", (t2-t1)/60, "min")
    return dictionary,corpus,lda

        df['tokenized'] = df['text'].apply(apply_all)
        all_words = [word for item in list(df['tokenized']) for word in item]
fdist = FreqDist(all_words)
k = 17000
top_k_words = fdist.most_common(k)
top_k_words[-10:]
top_k_words,_ = zip(*fdist.most_common(k))
top_k_words = set(top_k_words)
df['doc_len'] = df['tokenized'].apply(lambda x: len(x))
doc_lengths = list(df['doc_len'])
df.drop(labels='doc_len', axis=1, inplace=True)
num_bins = 1000
fig, ax = plt.subplots(figsize=(12,6));
# the histogram of the data
n, bins, patches = ax.hist(doc_lengths, num_bins, normed=1)
ax.set_xlabel('Document Length (tokens)', fontsize=15)
ax.set_ylabel('Normed Frequency', fontsize=15)
ax.grid()
ax.set_xticks(np.logspace(start=np.log10(50),stop=np.log10(2000),num=8, base=10.0))
plt.xlim(0,2000)
ax.plot([np.average(doc_lengths) for i in np.linspace(0.0,0.0035,100)], np.linspace(0.0,0.0035,100), '-',
        label='average doc length')
ax.legend()
ax.grid()
fig.tight_layout()
plt.show()
```

```
df = df[df['tokenized'].map(len) >= 30]
# make sure all tokenized items are lists
df = df[df['tokenized'].map(type) == list]
df.reset_index(drop=True,inplace=True)
msk = np.random.rand(len(df)) < 0.9
train_df = df[msk]
train_df.reset_index(drop=True,inplace=True)

test_df = df[~msk]
test_df.reset_index(drop=True,inplace=True)
#%%LDA train function
#%% Apply model

#dictionary,corpus,lda = train_lda(train_df)
#lda.save('lda_fakenews.model')
#dictionary.save('fakenews_dictionary')
dictionary = corpora.Dictionary.load('fakenews_dictionary')
lda=LdaModel.load('lda_fakenews.model')
# select and article at random from train_df
random_article_index = int(np.random.randint(len(train_df)))

bow = dictionary.doc2bow(train_df.iloc[random_article_index,5])
# get the topic contributions for the document chosen at random above
doc_distribution = np.array([tup[1] for tup in lda.get_document_topics(bow=bow)])
# bar plot of topic distribution for this document
fig, ax = plt.subplots(figsize=(12,6));
# the histogram of the data
patches = ax.bar(np.arange(len(doc_distribution)), doc_distribution)
ax.set_xlabel('Topic ID', fontsize=15)
ax.set_ylabel('Topic Contribution', fontsize=15)
ax.set_title("Topic Distribution for Article " + str(random_article_index), fontsize=20)
ax.set_xticks(np.linspace(10,100,10))
fig.tight_layout()
plt.show()

for i in doc_distribution.argsort()[-5:][::-1]:
    print(i, lda.show_topic(topicid=i, topn=10), "\n")
tm = test_df[test_df.label == 1]
dictionary_tm = corpora.Dictionary(tm['tokenized'])
new_bow = [dictionary.doc2bow(doc) for doc in tm['tokenized']]
new_doc_distribution_tm = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])
tp = test_df[test_df.label == 0]
dictionary_tp = corpora.Dictionary(tp['tokenized'])
new_bow = [dictionary.doc2bow(doc) for doc in tp['tokenized']]
new_doc_distribution_tp = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])
corpus_motor_train = [dictionary.doc2bow(doc) for doc in train_df['tokenized']]
doc_topic_dist = np.array([[tup[1] for tup in lst] for lst in lda[corpus_motor_train]])
doc_topic_dist.shape
```

```
all_sims_tm = []
for i in range(len(new_doc_distribution_tm)):
    doc_sims = jensen_shannon(new_doc_distribution_tm[i],doc_topic_dist)
    all_sims_tm.append(doc_sims)

tm = [item for sublist in all_sims_tm for item in sublist]
all_sims_tp = []
for i in range(len(new_doc_distribution_tp)):
    doc_sims = jensen_shannon(new_doc_distribution_tp[i],doc_topic_dist)
    all_sims_tp.append(doc_sims)
tp = [item for sublist in all_sims_tp for item in sublist]
plt.hist(tm, bins = 1000, lw = 0)
plt.show
plt.hist(tp, bins = 1000, lw = 0)
plt.show
tm.sort()
tp.sort()
cdf_tm= 1. * np.arange(len(tm)) / (len(tm) - 1)
cdf_tp= 1. * np.arange(len(tp)) / (len(tp) - 1)
tm = np.array(tm)
tp = np.array(tp)

plt.scatter(tm,cdf_tm, s = 0.5, color = 'g', label = 'Training motorcycles against test motorcycles')
plt.scatter(tp,cdf_tp, s = 0.5, color = 'r', label = 'Training motorcycles against politics')

plt.legend()
plt.show

from scipy import stats
x = stats.ks_2samp(tm, tp)
print(x)
```

Fake news with median LDA scores on test data-Relevance

```
import pandas as pd
import numpy as np
import nltk
from tqdm import tqdm
from nltk.corpus import stopwords
import gensim
from gensim.models import LdaModel
from gensim import models, corpora, similarities
import re
from scipy import stats
```

```python
from nltk.stem.porter import PorterStemmer
import time
from nltk import FreqDist
from scipy.stats import entropy
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import seaborn as sns
sns.set_style("darkgrid")
from sklearn.datasets import fetch_20newsgroups
from nltk.stem import WordNetLemmatizer


# ## Source of Data
# https://www.kaggle.com/c/fake-news/data

# In[2]:


#%% Download data
df = pd.read_csv('fake newstrain.csv')
df = df[df['text'].map(type) == str]
df['title'].fillna(value="", inplace=True)
df.dropna(axis=0, inplace=True, subset=['text'])
# shuffle the data
df = df.sample(frac=1.0)
df.reset_index(drop=True,inplace=True)


# In[3]:


def initial_clean(text):
    """
    Function to clean text of websites, email addresess and any punctuation
    We also lower case the text
    """
    text = re.sub('\s+', ' ', str(text))
    text = re.sub("((\S+)?(http(s)?)(\S+))|((\S+)?(www)(\S+))|((\S+)?(\@)(\S+)?)", " ", text)
    text = re.sub("[^a-zA-Z ]", "", text)
    text = text.lower() # lower case the text
    text = nltk.word_tokenize(text)
    return text

stop_words = stopwords.words('english')
def remove_stop_words(text):
    """
    Function that removes all stopwords from text
    """
```

```
    return [word for word in text if word not in stop_words]

lmtzr = WordNetLemmatizer()
def stem_words(text):
    """
    Function to stem words, so plural and singular are treated the same
    """

    text = [lmtzr.lemmatize(word) for word in text]
    text = [word for word in text if len(word) > 1] # make sure we have no 1 letter words
    return text

def apply_all(text):
    """
    This function applies all the functions above into one
    """
    return stem_words(remove_stop_words(initial_clean(text)))

def jensen_shannon(query, matrix):
    """
    This function implements a Jensen-Shannon similarity
    between the input query (an LDA topic distribution for a document)
    and the entire corpus of topic distributions.
    It returns an array of length M where M is the number of documents in the corpus
    """
    # lets keep with the p,q notation above
    p = query[None,:].T # take transpose
    q = matrix.T # transpose matrix
    m = 0.5*(p + q)
    return np.sqrt(0.5*(entropy(p,m) + entropy(q,m)))

def get_most_similar_documents(query,matrix,k=10):
    """
    This function implements the Jensen-Shannon distance above
    and retruns the top k indices of the smallest jensen shannon distances
    """
    sims = jensen_shannon(query,matrix) # list of jensen shannon distances
    return sims.argsort()[:k] # the top k positional index of the smallest Jensen Shannon distances


# In[4]:


df['tokenized'] = df['text'].apply(apply_all)


# In[5]:

```

```

# only keep articles with more than 30 tokens, otherwise too short
df = df[df['tokenized'].map(len) >= 40]
# make sure all tokenized items are lists
df = df[df['tokenized'].map(type) == list]
df.reset_index(drop=True,inplace=True)
print("After cleaning and excluding short aticles, the dataframe now has:", len(df), "articles")


# We decide to use the the training data as both training and test as we have easily accessible labelled data

# In[6]:


msk = np.random.rand(len(df)) < 0.9
train_df = df[msk]
train_df.reset_index(drop=True,inplace=True)

test_df = df[~msk]
test_df.reset_index(drop=True,inplace=True)


# In[7]:


train_df.shape


# ## Load saved model and dictionary

# In[8]:


dictionary = corpora.Dictionary.load('fakenews_dictionary_clean')


# In[9]:


lda=LdaModel.load('lda_fakenews_clean.mode')


# We decide to to test our models performance on how well it can discriminate between reliable and unreliable data

# label: a label that marks the article as potentially unreliable
# 1: unreliable
# 0: reliable

```

```
# In[10]:


#%%tUnreliable data
tm = test_df[test_df.label == 1]


# In[11]:


dictionary_tm = corpora.Dictionary(tm['tokenized'])
new_bow = [dictionary.doc2bow(doc) for doc in tm['tokenized']]
new_doc_distribution_tm = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])


# In[12]:


#Reliable data
tp = test_df[test_df.label == 0]


# In[13]:


#%%
dictionary_tp = corpora.Dictionary(tp['tokenized'])
new_bow = [dictionary.doc2bow(doc) for doc in tp['tokenized']]
new_doc_distribution_tp = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])


# In[14]:


corpus_motor_train = [dictionary.doc2bow(doc) for doc in train_df['tokenized']]


# In[15]:


doc_topic_dist = np.array([[tup[1] for tup in lst] for lst in lda[corpus_motor_train]])
doc_topic_dist.shape


# In[ ]:
```

```
# In[33]:


all_sims_tm = []
for i in tqdm(range(len(new_doc_distribution_tm))):
    doc_sims = jensen_shannon(new_doc_distribution_tm[i],doc_topic_dist)
    all_sims_tm.append(1-np.median(doc_sims))


# In[34]:


all_sims_tm=[x for x in all_sims_tm if str(x) != 'nan']


# In[35]:


#%% this is surprisingly fast
#most_sim_ids = get_most_similar_documents(new_doc_distribution,doc_topic_dist)
all_sims_tp = []
for i in tqdm(range(len(new_doc_distribution_tp))):
    doc_sims = jensen_shannon(new_doc_distribution_tp[i],doc_topic_dist)
    all_sims_tp.append(1-np.median(doc_sims))
#most_similar_df['title']


# In[36]:


all_sims_tp=[x for x in all_sims_tp if str(x) != 'nan']


# In[37]:


all_sims=all_sims_tm+all_sims_tp


# In[38]:


tm_label=['r']*len(all_sims_tm)


# In[39]:
```

```python
tp_label=['b']*len(all_sims_tp)


# In[40]:


labels=tm_label+tp_label


# In[41]:


len(labels)


# In[42]:


# create dataframe of rel_index and label
df_original = pd.DataFrame({'label': labels,
                            'rel_index': all_sims})

df = df_original.sort_values('rel_index', ascending=False)
# find threshold (minimum relevance index of relevant articles)
threshold = df.loc[df['label'] == 'b']['rel_index'].min()

# true relevant (relevant articles above threshold)
true_relevant = df.loc[(df["label"] == 'b') & (df["rel_index"] > threshold)]
false_relevant = df.loc[(df["label"] == 'b') & (df["rel_index"] < threshold)]

# true irrelevant (irrelevant articles below threshold)
true_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] < threshold)]
false_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] > threshold)]

perc_ignore = float(len(true_irrelevant)) / (len(df)) * 100
precision = float(len(true_irrelevant)) / float(len(false_irrelevant)+len(true_irrelevant))
accuracy=(len(true_relevant)+len(true_irrelevant))/ len(df)
try:
    recall=float(len(true_irrelevant)) / float(len(false_relevant)+len(true_irrelevant))
except ZeroDivisionError:
    recall =0
    # false irrelevant (irrelevant articles above threshold)
false_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] >= threshold)]

plt.figure(figsize=(25, 10))
plt.ylim([-1,max(df['rel_index']+1)])
```

```
plt.xlim([0, len(labels)])
plt.scatter(range(len(df)), df['rel_index'], c=df['label'], s=30,alpha=0.7)
# plot threshold
plt.axhline(threshold, c='black', linewidth=1.5)
plt.ylabel('relevance index')
plt.text(1000, 1.0, r'True irrelevant: ' + str(len(true_irrelevant)) + '\n' + 'From total of: ' + str(
        len(df)) + '  (' + "%.2f" % perc_ignore + '%)', verticalalignment='bottom', horizontalalignment='left')


#
```

Fake news with median LDA scores on test data relevance plot

```
#!/usr/bin/env python
# coding: utf-8

# In[5]:


import pandas as pd
import numpy as np
import nltk
from tqdm import tqdm
from nltk.corpus import stopwords
import gensim
from gensim.models import LdaModel
from gensim import models, corpora, similarities
import re
from scipy import stats
from nltk.stem.porter import PorterStemmer
import time
from nltk import FreqDist
from scipy.stats import entropy
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import seaborn as sns
sns.set_style("darkgrid")
from sklearn.datasets import fetch_20newsgroups
from nltk.stem import WordNetLemmatizer


# ## Source of Data
# https://www.kaggle.com/c/fake-news/data

```

```python
# In[2]:


#%% Download data
df = pd.read_csv('fake newstrain.csv')
df = df[df['text'].map(type) == str]
df['title'].fillna(value="", inplace=True)
df.dropna(axis=0, inplace=True, subset=['text'])
# shuffle the data
df = df.sample(frac=1.0)
df.reset_index(drop=True,inplace=True)
df.head()


# In[3]:


def initial_clean(text):
    """
    Function to clean text of websites, email addresess and any punctuation
    We also lower case the text
    """
    text = re.sub('\s+', ' ', str(text))
    text = re.sub("((\S+)?(http(s)?)(\S+))|((\S+)?(www)(\S+))|((\S+)?(\@)(\S+)?)", " ", text)
    text = re.sub("[^a-zA-Z ]", "", text)
    text = text.lower() # lower case the text
    text = nltk.word_tokenize(text)
    return text

stop_words = stopwords.words('english')
def remove_stop_words(text):
    """
    Function that removes all stopwords from text
    """
    return [word for word in text if word not in stop_words]

lmtzr = WordNetLemmatizer()
def stem_words(text):
    """
    Function to stem words, so plural and singular are treated the same
    """

    text = [lmtzr.lemmatize(word) for word in text]
    text = [word for word in text if len(word) > 1] # make sure we have no 1 letter words
    return text

def apply_all(text):
    """
```

```python
    This function applies all the functions above into one
    """
    return stem_words(remove_stop_words(initial_clean(text)))

def jensen_shannon(query, matrix):
    """
    This function implements a Jensen-Shannon similarity
    between the input query (an LDA topic distribution for a document)
    and the entire corpus of topic distributions.
    It returns an array of length M where M is the number of documents in the corpus
    """
    # lets keep with the p,q notation above
    p = query[None,:].T # take transpose
    q = matrix.T # transpose matrix
    m = 0.5*(p + q)
    return np.sqrt(0.5*(entropy(p,m) + entropy(q,m)))

def get_most_similar_documents(query,matrix,k=10):
    """
    This function implements the Jensen-Shannon distance above
    and retruns the top k indices of the smallest jensen shannon distances
    """
    sims = jensen_shannon(query,matrix) # list of jensen shannon distances
    return sims.argsort()[:k] # the top k positional index of the smallest Jensen Shannon distances

def keep_top_k_words(text):
    return [word for word in text if word in top_k_words]

def train_lda(data):
    """
    This function trains the lda model
    We setup parameters like number of topics, the chunksize to use in Hoffman method
    We also do 2 passes of the data since this is a small dataset, so we want the distributions to stabilize
    """
    num_topics = 100
    chunksize = 300
    dictionary = corpora.Dictionary(data['tokenized'])
    corpus = [dictionary.doc2bow(doc) for doc in data['tokenized']]
    t1 = time.time()
    # low alpha means each document is only represented by a small number of topics, and vice versa
    # low eta means each topic is only represented by a small number of words, and vice versa
    lda = LdaModel(corpus=corpus, num_topics=num_topics, id2word=dictionary,
                   alpha=1e-2, eta=0.5e-2, chunksize=chunksize, minimum_probability=0.0, passes=2)
    t2 = time.time()
    print("Time to train LDA model on ", len(df), "articles: ", (t2-t1)/60, "min")
    return dictionary,corpus,lda


```

```


# In[4]:


df['tokenized'] = df['text'].apply(apply_all)


# In[6]:


# only keep articles with more than 30 tokens, otherwise too short
df = df[df['tokenized'].map(len) >= 40]
# make sure all tokenized items are lists
df = df[df['tokenized'].map(type) == list]
df.reset_index(drop=True,inplace=True)
print("After cleaning and excluding short aticles, the dataframe now has:", len(df), "articles")


# We decide to use the the training data as both training and test as we have easily accessible labelled data

# In[7]:


msk = np.random.rand(len(df)) < 0.9
train_df = df[msk]
train_df.reset_index(drop=True,inplace=True)

test_df = df[~msk]
test_df.reset_index(drop=True,inplace=True)


# In[8]:


train_df.shape


# ## Load saved model and dictionary

# In[9]:


dictionary = corpora.Dictionary.load('fakenews_dictionary_clean')


# In[10]:

```

```
lda=LdaModel.load('lda_fakenews_clean.mode')


# We decide to to test our models performance on how well it can discriminate between reliable and unreliable data

# label: a label that marks the article as potentially unreliable
# 1: unreliable
# 0: reliable

# In[11]:


#%%tUnreliable data
tm = test_df[test_df.label == 1]


# In[12]:


dictionary_tm = corpora.Dictionary(tm['tokenized'])
new_bow = [dictionary.doc2bow(doc) for doc in tm['tokenized']]
new_doc_distribution_tm = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])


# In[13]:


#Reliable data
tp = test_df[test_df.label == 0]


# In[14]:


#%%
dictionary_tp = corpora.Dictionary(tp['tokenized'])
new_bow = [dictionary.doc2bow(doc) for doc in tp['tokenized']]
new_doc_distribution_tp = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])


# In[15]:


corpus_motor_train = [dictionary.doc2bow(doc) for doc in train_df['tokenized']]


# In[16]:
```

```
doc_topic_dist = np.array([[tup[1] for tup in lst] for lst in lda[corpus_motor_train]])
doc_topic_dist.shape


# In[18]:





# In[101]:


all_sims_tm = []
for i in tqdm(range(len(new_doc_distribution_tm))):
    doc_sims = jensen_shannon(new_doc_distribution_tm[i],doc_topic_dist)
    all_sims_tm.append(np.median(doc_sims))


# In[111]:


all_sims_tm=[x for x in all_sims_tm if str(x) != 'nan']


# In[100]:


#%% this is surprisingly fast
#most_sim_ids = get_most_similar_documents(new_doc_distribution,doc_topic_dist)
all_sims_tp = []
for i in tqdm(range(len(new_doc_distribution_tp))):
    doc_sims = jensen_shannon(new_doc_distribution_tp[i],doc_topic_dist)
    all_sims_tp.append(np.median(doc_sims))
#most_similar_df['title']


# In[102]:


all_sims_tp=[x for x in all_sims_tp if str(x) != 'nan']


# In[104]:
```

```
get_ipython().run_line_magic('matplotlib', 'inline')


# In[112]:


all_sims_tm.sort()
all_sims_tp.sort()


# In[113]:


cdf_tm= 1. * np.arange(len(all_sims_tm)) / (len(all_sims_tm) - 1)
cdf_tp= 1. * np.arange(len(all_sims_tp)) / (len(all_sims_tp) - 1)


# In[145]:


plt.title('Reliable v Unreliable CDF distance distributions')
plt.scatter(all_sims_tm,cdf_tm,s=0.7, color = 'r', label = 'Lda model on unreliable test data ')
plt.scatter(all_sims_tp,cdf_tp,s=0.7, color = 'g', label = 'LDA model on reliable test data')
plt.legend()
plt.show


# KS intepretation
# https://towardsdatascience.com/kolmogorov-smirnov-test-84c92fb4158d

# In[118]:


x = stats.ks_2samp(all_sims_tm, all_sims_tp)


# In[119]:


print(x)


# In[120]:


x.statistic


```

```python
# In[121]:


x.pvalue


# In[128]:


all_sims_tm = np.array(all_sims_tm)
all_sims_tp = np.array(all_sims_tp)


# For intepretation of the Anderson-Darling test.
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.anderson_ksamp.html

# In[136]:


y=stats.anderson_ksamp([all_sims_tp,all_sims_tm])


# In[137]:


print(y)


# In[146]:


z=stats.ttest_ind(all_sims_tp,all_sims_tm,equal_var = False)


# In[147]:


print(z)
```

Word2vec Fakenews

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:
```

```python


import pandas as pd
import numpy as np
import gensim
import time
import re
import nltk
import matplotlib.pyplot as plt
import seaborn as sns
from nltk.corpus import stopwords
sns.set_style("darkgrid")
from sklearn.datasets import fetch_20newsgroups
from nltk.stem import WordNetLemmatizer
from gensim.models import Word2Vec,KeyedVectors
from gensim.test.utils import common_texts
from gensim.corpora import Dictionary
from gensim.models import Word2Vec, WordEmbeddingSimilarityIndex
from gensim.similarities import SoftCosineSimilarity, SparseTermSimilarityMatrix
import seaborn as sns
from gensim.corpora import Dictionary
import gensim
# upgrade gensim if you can't import softcossim
from gensim.matutils import softcossim
from gensim import corpora
from gensim.utils import simple_preprocess


# In[2]:


import logging  # Setting up the loggings to monitor gensim
logging.basicConfig(format="%(levelname)s - %(asctime)s: %(message)s", datefmt= '%H:%M:%S', level=logging.INFO)


# In[3]:


def initial_clean(text):
    """
    Function to clean text of websites, email addresess and any punctuation
    We also lower case the text
    """
    text = re.sub('\s+', ' ', str(text))
    text = re.sub("((\S+)?(http(s)?)(\S+))|((\S+)?(www)(\S+))|((\S+)?(\@)(\S+)?)", " ", text)
    text = re.sub("[^a-zA-Z ]", "", text)
    text = text.lower() # lower case the text
    text = nltk.word_tokenize(text)
```

```
    return text


# In[4]:


#%% Download data
df = pd.read_csv('fake newstrain.csv')
df = df[df['text'].map(type) == str]
df['title'].fillna(value="", inplace=True)


# In[5]:


df.isnull().sum()


# In[6]:


df.dropna(axis=0, inplace=True, subset=['text'])
# shuffle the data
df = df.sample(frac=1.0)
df.reset_index(drop=True,inplace=True)


# In[7]:


df.isnull().sum()


# In[8]:


df['tokenized'] = df['text'].apply(initial_clean)


# In[9]:


# only keep articles with more than 30 tokens, otherwise too short
df = df[df['tokenized'].map(len) >= 2]
# changed from 30 to 2 for word2vec as a result of https://www.kaggle.com/pierremegret/gensim-word2vec-tutorial
# make sure all tokenized items are lists
```

```python
# In[10]:


df = df[df['tokenized'].map(type) == list]
df.reset_index(drop=True,inplace=True)
print("After cleaning and excluding short aticles, the dataframe now has:", len(df), "articles")


# In[11]:


df['tokenized'] = df['tokenized'].apply(' '.join)


# In[12]:


df.dropna(axis=0, inplace=True, subset=['tokenized'])


# In[13]:


msk = np.random.rand(len(df)) < 0.9
train_df = df[msk]
train_df.reset_index(drop=True,inplace=True)

test_df = df[~msk]
test_df.reset_index(drop=True,inplace=True)


# In[14]:


#%%word2vec train function
def train_w2v(data):
    """
    This function trains the word2vec model
    """
    t1 = time.time()
    word2vec = Word2Vec(sentences=data, size=len(data), window=5, min_count=5, workers=4, sg=0)
    #index the words
    index2word_set = list(word2vec.wv.index2word)
    word_vectors = word2vec.wv #all information is stored in word vectors
    del word2vec #delete model to trim unneeded model state

    t2 = time.time()
    print("Time to train word2vec model on ", len(df), "articles: ", (t2-t1)/60, "min")
```

```python
    return word_vectors, index2word_set

#%%cossim function
def create_soft_cossim_matrix(docsim_index, query_corpus, dictionary):
    all_soft_cossim = []
    for i in query_corpus:
        if len(i) > 0:
            sims = list(docsim_index[i])
            all_soft_cossim.extend(sims)
    return all_soft_cossim


# In[15]:


#word2vec = Word2Vec(sentences=df['tokenized'], size=300,window=5, min_count=5, workers=4, sg=0)


# In[16]:


#word2vec.init_sims(replace=True)


# In[17]:


#word_vectors=KeyedVectors.load('w2v fake_news')


# In[18]:



"""
#%% Apply model
word_vectors, index2word_set = train_w2v([apply_all(doc) for doc in df['text']])
#%%
termsim_index = WordEmbeddingSimilarityIndex(word_vectors)
#To compute soft cosines, you need the dictionary (a map of word to unique id),
#the corpus (word counts) for each sentence and the similarity matrix

# Prepare a dictionary and a corpus.
dictionary = corpora.Dictionary([simple_preprocess(doc) for doc in df['Data']])

# Prepare the similarity matrix
similarity_matrix = SparseTermSimilarityMatrix(termsim_index, dictionary)  # construct similarity matrix
"""
```

```
# In[19]:


#word_vectors.save('w2v fake_news')
#dictionary.save('w2v fake news dictionary')
#similarity_matrix.save("w2v fakenews similarity")


# In[20]:


len(df['tokenized'])


# In[21]:


word_vectors=KeyedVectors.load('w2v fake_news')


# In[22]:


#word_vectors = word2vec.wv


# In[23]:


#word_vectors=KeyedVectors.load('word2vec.model')


# In[24]:


len(word_vectors.wv.vocab)


# In[25]:


termsim_index = WordEmbeddingSimilarityIndex(word_vectors)


# In[26]:

```

```
dictionary = corpora.Dictionary.load('w2v fake news dictionary')


# In[27]:


#dictionary = corpora.Dictionary([simple_preprocess(doc) for doc in df['tokenized']])


# In[28]:


similarity_matrix = SparseTermSimilarityMatrix(termsim_index, dictionary)  # construct similarity matrix


# ### train_set we get the softcosine similarity for one set

# In[73]:


#Now we are ready to calculate softcossim.
#What we testing against (Convert the train docs into bow)
#%%
train_set = [dictionary.doc2bow(simple_preprocess(doc)) for doc in train_df.loc[train_df.label == 1]['tokenized']
train_set2 = [dictionary.doc2bow(simple_preprocess(doc)) for doc in train_df.loc[train_df.label == 0]['tokenized'
train_set = train_set+train_set2
docsim_index = SoftCosineSimilarity(train_set, similarity_matrix)


# In[74]:


len(train_set)


# In[32]:


tm = [dictionary.doc2bow(simple_preprocess(doc)) for doc in test_df[test_df.label == 1]['tokenized']]
tp = [dictionary.doc2bow(simple_preprocess(doc)) for doc in test_df[test_df.label == 0]['tokenized']]


# In[37]:


test_doc=tm+tp
```

```
# In[38]:


unrel_labels=['r']*len(tm)


# In[39]:


rel_labels=['b']*len(tm)


# In[40]:


labels= unrel_labels+rel_labels


# In[69]:


len(labels)


# In[43]:


def chunks(l, n):
    # For item i in a range that is a length of l,
    for i in range(0, len(l), n):
        # Create an index range for l of n items:
        yield l[i:i+n]


# In[44]:


import statistics
import scipy.stats


# In[45]:



def mean_confidence_interval(data, confidence=0.95):
    a = 1.0 * np.array(data)
```

```
    n = len(a)
    m, se = np.mean(a), scipy.stats.sem(a)
    h = se * scipy.stats.t.ppf((1 + confidence) / 2., n-1)
    return m, m-h, m+h


# In[78]:


def create_soft_cossim_median_matrix(docsim_index, query_corpus, dictionary):
    statslist = []
    for i in query_corpus:
        if len(i) > 0:
            sims = docsim_index[i]
            sims_array=np.array(sims)
            med=np.median(sims_array)
            statslist.extend([med])
    return statslist


# In[76]:


def create_soft_cossim_ci_matrix(docsim_index, query_corpus, dictionary):
    statslist = []
    for i in query_corpus:
        print(i)
        if len(i) > 0:
            sims = docsim_index[i]
            mean,lu,up=mean_confidence_interval(sims)
            median=np.median(sims)
            stats=[lu,median,mean,up]
            statslist.append([stats])
    return statslist


# In[52]:


tm_chunks=list(chunks(tm,300))


# In[53]:


tp_chunks=list(chunks(tp,300))
```

```
# In[47]:


for i in tm_chunks:
    all_sims_tm = create_soft_cossim_matrix(docsim_index, i, dictionary)


# In[48]:


for i in tp_chunks:
    all_sims_tp = create_soft_cossim_matrix(docsim_index, i, dictionary)


# In[79]:


all_sims = create_soft_cossim_median_matrix(docsim_index, test_doc, dictionary)


# In[80]:


len(all_sims)


# In[81]:


all_sims=all_sims[:2050]


# In[49]:


#%% cdf
all_sims_tm.sort()
all_sims_tp.sort()


# In[50]:


cdf_tm= 1. * np.arange(len(all_sims_tm)) / (len(all_sims_tm) - 1)
cdf_tp= 1. * np.arange(len(all_sims_tp)) / (len(all_sims_tp) - 1)


# In[51]:
```

```python


all_sims_tm_array = np.array(all_sims_tm)
all_sims_tp_array = np.array(all_sims_tp)


# In[63]:


plt.scatter(all_sims_tm_array,cdf_tm, s = 0.5, color = 'green', label = 'Word2vec on unreliable test data ')
plt.scatter(all_sims_tp_array,cdf_tp, s = 0.5, color = 'red', label = 'Word2vec on reliable test data')
plt.title("Word2Vec similarity curve")


plt.legend()
plt.show


# In[53]:


from scipy import stats


# In[56]:


x = stats.ks_2samp(all_sims_tm_array, all_sims_tp_array)


# In[57]:


print(x)


# In[58]:


x.statistic


# In[20]:


x.pvalue


```

```
# In[59]:


y=stats.anderson_ksamp([all_sims_tp,all_sims_tm])


# In[60]:


y


# In[61]:


z=stats.ttest_ind(all_sims_tp,all_sims_tm,equal_var = False)


# In[62]:


z


# In[82]:


len(labels)


# In[83]:


len(all_sims)


# In[84]:


# create dataframe of rel_index and label
df_original = pd.DataFrame({'label': labels,
                            'rel_index': all_sims})


# In[85]:


df = df_original.sort_values('rel_index', ascending=False)
```

```
# In[110]:


# find threshold (minimum relevance index of relevant articles)
threshold = df.loc[df['label'] == 'b']['rel_index'].min()

# true relevant (relevant articles above threshold)
true_relevant = df.loc[(df["label"] == 'b') & (df["rel_index"] > threshold)]
false_relevant = df.loc[(df["label"] == 'b') & (df["rel_index"] < threshold)]

# true irrelevant (irrelevant articles below threshold)
true_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] < threshold)]
false_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] > threshold)]

perc_ignore = float(len(true_irrelevant)) / (len(df)) * 100
precision = float(len(true_irrelevant)) / float(len(false_irrelevant)+len(true_irrelevant))
accuracy=(len(true_relevant)+len(true_irrelevant))/ len(df)
recall=float(len(true_irrelevant)) / float(len(false_relevant)+len(true_irrelevant))
# false irrelevant (irrelevant articles above threshold)
false_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] >= threshold)]

plt.figure(figsize=(25, 10))
plt.ylim([-1,max(df['rel_index']+1)])
plt.xlim([0, len(labels)])
plt.scatter(range(len(df)), df['rel_index'], c=df['label'], s=30,alpha=0.7)
# plot threshold
plt.axhline(threshold, c='black', linewidth=1.5)
plt.title(
        'Case Study: capeunimart' + '\n' +
        'Blue - relevant, Red - irrelevant' + '\n' +
        'Frame length: 20, 50 topics, metric - max')
plt.ylabel('relevance index')
plt.text(1000, 1.0, r'True irrelevant: ' + str(len(true_irrelevant)) + '\n' + 'From total of: ' + str(
        len(df)) + '  (' + "%.2f" % perc_ignore + '%)', verticalalignment='bottom', horizontalalignment='left')


# In[94]:


precision


# In[95]:


recall
```

```
# In[96]:


accuracy
```

Word2vec Fakenews-Relevance Plot

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import pandas as pd
import numpy as np
import gensim
import time
import re
import nltk
import matplotlib.pyplot as plt
import seaborn as sns
from nltk.corpus import stopwords
sns.set_style("darkgrid")
from sklearn.datasets import fetch_20newsgroups
from nltk.stem import WordNetLemmatizer
from gensim.models import Word2Vec,KeyedVectors
from gensim.test.utils import common_texts
from gensim.corpora import Dictionary
from gensim.models import Word2Vec, WordEmbeddingSimilarityIndex
from gensim.similarities import SoftCosineSimilarity, SparseTermSimilarityMatrix
import seaborn as sns
from gensim.corpora import Dictionary
import gensim
# upgrade gensim if you can't import softcossim
from gensim.matutils import softcossim
from gensim import corpora
from gensim.utils import simple_preprocess


# In[2]:


import logging  # Setting up the loggings to monitor gensim
```

```
logging.basicConfig(format="%(levelname)s - %(asctime)s: %(message)s", datefmt= '%H:%M:%S', level=logging.INFO)


# In[3]:


def initial_clean(text):
    """
    Function to clean text of websites, email addresess and any punctuation
    We also lower case the text
    """
    text = re.sub('\s+', ' ', str(text))
    text = re.sub("((\S+)?(http(s)?)(\S+))|((\S+)?(www)(\S+))|((\S+)?(\@)(\S+)?)", " ", text)
    text = re.sub("[^a-zA-Z ]", "", text)
    text = text.lower() # lower case the text
    text = nltk.word_tokenize(text)
    return text


# In[4]:


#%% Download data
df = pd.read_csv('fake newstrain.csv')
df = df[df['text'].map(type) == str]
df['title'].fillna(value="", inplace=True)


# In[5]:


df.isnull().sum()


# In[6]:


df.dropna(axis=0, inplace=True, subset=['text'])
# shuffle the data
df = df.sample(frac=1.0)
df.reset_index(drop=True,inplace=True)


# In[7]:


df.isnull().sum()

```

```python

# In[8]:


df['tokenized'] = df['text'].apply(initial_clean)


# In[9]:


# only keep articles with more than 30 tokens, otherwise too short
df = df[df['tokenized'].map(len) >= 2]
# changed from 30 to 2 for word2vec as a result of https://www.kaggle.com/pierremegret/gensim-word2vec-tutorial
# make sure all tokenized items are lists


# In[10]:


df = df[df['tokenized'].map(type) == list]
df.reset_index(drop=True,inplace=True)
print("After cleaning and excluding short aticles, the dataframe now has:", len(df), "articles")


# In[11]:


df['tokenized'] = df['tokenized'].apply(' '.join)


# In[12]:


df.dropna(axis=0, inplace=True, subset=['tokenized'])


# In[13]:


msk = np.random.rand(len(df)) < 0.9
train_df = df[msk]
train_df.reset_index(drop=True,inplace=True)

test_df = df[~msk]
test_df.reset_index(drop=True,inplace=True)


# In[14]:
```

```
word_vectors=KeyedVectors.load('w2v fake_news')


# In[15]:


len(word_vectors.wv.vocab)


# In[16]:


termsim_index = WordEmbeddingSimilarityIndex(word_vectors)


# In[17]:


dictionary = corpora.Dictionary.load('w2v fake news dictionary')


# In[18]:


#dictionary = corpora.Dictionary([simple_preprocess(doc) for doc in df['tokenized']])


# In[19]:


similarity_matrix = SparseTermSimilarityMatrix(termsim_index, dictionary)  # construct similarity matrix


# ### train_set we get the softcosine similarity for one set

# In[20]:


#Now we are ready to calculate softcossim.
#What we testing against (Convert the train docs into bow)
#%%
train_set = [dictionary.doc2bow(simple_preprocess(doc)) for doc in train_df['tokenized']]
docsim_index = SoftCosineSimilarity(train_set, similarity_matrix)


# In[21]:
```

```


tm = [dictionary.doc2bow(simple_preprocess(doc)) for doc in test_df[test_df.label == 1]['tokenized']]
tp = [dictionary.doc2bow(simple_preprocess(doc)) for doc in test_df[test_df.label == 0]['tokenized']]


# In[23]:


def chunks(l, n):
    # For item i in a range that is a length of l,
    for i in range(0, len(l), n):
        # Create an index range for l of n items:
        yield l[i:i+n]


# In[24]:


import statistics
import scipy.stats


# In[25]:



def mean_confidence_interval(data, confidence=0.95):
    a = 1.0 * np.array(data)
    n = len(a)
    m, se = np.mean(a), scipy.stats.sem(a)
    h = se * scipy.stats.t.ppf((1 + confidence) / 2., n-1)
    return m, m-h, m+h


# In[26]:


def create_soft_cossim_median_matrix(docsim_index, query_corpus, dictionary):
    statslist = []
    for i in query_corpus:
        if len(i) > 0:
            sims = docsim_index[i]
            sims_array=np.array(sims)
            med=np.median(sims_array)
            statslist.extend([med])
    return statslist

```

```python

# In[27]:


def create_soft_cossim_ci_matrix(docsim_index, query_corpus, dictionary):
    statslist = []
    for i in query_corpus:
        print(i)
        if len(i) > 0:
            sims = docsim_index[i]
            mean,lu,up=mean_confidence_interval(sims)
            median=np.median(sims)
            stats=[lu,median,mean,up]
            statslist.append([stats])
    return statslist


# In[28]:


tm_chunks=list(chunks(tm,300))


# In[29]:


tp_chunks=list(chunks(tp,300))


# In[30]:


for i in tm_chunks:
    all_sims_tm = create_soft_cossim_median_matrix(docsim_index, i, dictionary)


# In[31]:


for i in tp_chunks:
    all_sims_tp = create_soft_cossim_median_matrix(docsim_index, i, dictionary)


# In[32]:


unrel_labels=['r']*len(all_sims_tm)

```

```
# In[33]:


rel_labels=['b']*len(all_sims_tp)


# In[34]:


labels= unrel_labels+rel_labels


# In[35]:


all_sims = all_sims_tm+all_sims_tp


# In[36]:


# create dataframe of rel_index and label
df_original = pd.DataFrame({'label': labels,
                            'rel_index': all_sims})


# In[37]:


df = df_original.sort_values('rel_index', ascending=False)


# In[38]:


# find threshold (minimum relevance index of relevant articles)
threshold = df.loc[df['label'] == 'b']['rel_index'].min()

# true relevant (relevant articles above threshold)
true_relevant = df.loc[(df["label"] == 'b') & (df["rel_index"] > threshold)]
false_relevant = df.loc[(df["label"] == 'b') & (df["rel_index"] < threshold)]

# true irrelevant (irrelevant articles below threshold)
true_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] < threshold)]
false_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] > threshold)]

perc_ignore = float(len(true_irrelevant)) / (len(df)) * 100
```

```
precision = float(len(true_irrelevant)) / float(len(false_irrelevant)+len(true_irrelevant))
accuracy=(len(true_relevant)+len(true_irrelevant))/ len(df)
recall=float(len(true_irrelevant)) / float(len(false_relevant)+len(true_irrelevant))
# false irrelevant (irrelevant articles above threshold)
false_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] >= threshold)]

plt.figure(figsize=(25, 10))
plt.ylim([-1,max(df['rel_index']+1)])
plt.xlim([0, len(labels)])
plt.scatter(range(len(df)), df['rel_index'], c=df['label'], s=30,alpha=0.7)
# plot threshold
plt.axhline(threshold, c='black', linewidth=1.5)
plt.title(
        'Case Study: capeunimart' + '\n' +
        'Blue - relevant, Red - irrelevant' + '\n' +
        'Frame length: 20, 50 topics, metric - max')
plt.ylabel('relevance index')
plt.text(1000, 1.0, r'True irrelevant: ' + str(len(true_irrelevant)) + '\n' + 'From total of: ' + str(
        len(df)) + '  (' + "%.2f" % perc_ignore + '%)', verticalalignment='bottom', horizontalalignment='left')


# In[94]:


precision


# In[95]:


recall


# In[96]:


accuracy


# In[ ]:
```

Motorcycle/Auto sports Similarity Analysis for LDA

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import pandas as pd
import numpy as np
import nltk
from nltk.corpus import stopwords
import gensim
from gensim.models import LdaModel
from gensim import models, corpora, similarities
import re
from nltk.stem.porter import PorterStemmer
import time
from nltk import FreqDist
from scipy.stats import entropy
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")
from sklearn.datasets import fetch_20newsgroups
from nltk.stem import WordNetLemmatizer


# In[2]:


#%% Download data
newsgroups_train = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'))
sections = list(newsgroups_train.target_names)


df = pd.DataFrame(columns = ['Type', 'Data'])

data = []
names = []
for i in range(len(sections)):
    d = fetch_20newsgroups(shuffle=True, random_state=1,subset = 'train', remove=('headers', 'footers', 'quotes')
    data.append(d.data)
    n = [sections[i]]*len(d.data)
    names.append(n)

```

```python

# In[3]:


together_data = [item for sublist in data for item in sublist]
together_name = [item for sublist in names for item in sublist]

df = df.assign(Type = together_name, Data = together_data)


# In[4]:


def initial_clean(text):
    """
    Function to clean text of websites, email addresess and any punctuation
    We also lower case the text
    """
    text = re.sub('\s+', ' ', text)
    text = re.sub("((\S+)?(http(s)?)(\S+))|((\S+)?(www)(\S+))|((\S+)?(\@)(\S+)?)", " ", text)
    text = re.sub("[^a-zA-Z ]", "", text)
    text = text.lower() # lower case the text
    text = nltk.word_tokenize(text)
    return text

stop_words = stopwords.words('english')
def remove_stop_words(text):
    """
    Function that removes all stopwords from text
    """
    return [word for word in text if word not in stop_words]

lmtzr = WordNetLemmatizer()
def stem_words(text):
    """
    Function to stem words, so plural and singular are treated the same
    """

    text = [lmtzr.lemmatize(word) for word in text]
    text = [word for word in text if len(word) > 1] # make sure we have no 1 letter words
    return text

def apply_all(text):
    """
    This function applies all the functions above into one
    """
    return stem_words(remove_stop_words(initial_clean(text)))

```

```
def jensen_shannon(query, matrix):
    """
    This function implements a Jensen-Shannon similarity
    between the input query (an LDA topic distribution for a document)
    and the entire corpus of topic distributions.
    It returns an array of length M where M is the number of documents in the corpus
    """
    # lets keep with the p,q notation above
    p = query[None,:].T # take transpose
    q = matrix.T # transpose matrix
    m = 0.5*(p + q)
    return np.sqrt(0.5*(entropy(p,m) + entropy(q,m)))

def get_most_similar_documents(query,matrix,k=10):
    """
    This function implements the Jensen-Shannon distance above
    and retruns the top k indices of the smallest jensen shannon distances
    """
    sims = jensen_shannon(query,matrix) # list of jensen shannon distances
    return sims.argsort()[:k] # the top k positional index of the smallest Jensen Shannon distances

def train_lda(data):
    """
    This function trains the lda model
    We setup parameters like number of topics, the chunksize to use in Hoffman method
    We also do 2 passes of the data since this is a small dataset, so we want the distributions to stabilize
    """
    num_topics = 100
    chunksize = 300
    dictionary = corpora.Dictionary(data['tokenized'])
    corpus = [dictionary.doc2bow(doc) for doc in data['tokenized']]
    t1 = time.time()
    # low alpha means each document is only represented by a small number of topics, and vice versa
    # low eta means each topic is only represented by a small number of words, and vice versa
    lda = LdaModel(corpus=corpus, num_topics=num_topics, id2word=dictionary,
                   alpha=1e-2, eta=0.5e-2, chunksize=chunksize, minimum_probability=0.0, passes=2)
    t2 = time.time()
    print("Time to train LDA model on ", len(df), "articles: ", (t2-t1)/60, "min")
    return dictionary,corpus,lda


# In[5]:


df['tokenized'] = df['Data'].apply(apply_all)


# In[10]:
```

```
print("length of list:",len(doc_lengths),
      "\naverage document length", np.average(doc_lengths),
      "\nminimum document length", min(doc_lengths),
      "\nmaximum document length", max(doc_lengths))


# In[11]:


# plot a histogram of document length
num_bins = 1000
fig, ax = plt.subplots(figsize=(12,6));
# the histogram of the data
n, bins, patches = ax.hist(doc_lengths, num_bins, normed=1)
ax.set_xlabel('Document Length (tokens)', fontsize=15)
ax.set_ylabel('Normed Frequency', fontsize=15)
ax.grid()
ax.set_xticks(np.logspace(start=np.log10(50),stop=np.log10(2000),num=8, base=10.0))
plt.xlim(0,2000)
ax.plot([np.average(doc_lengths) for i in np.linspace(0.0,0.0035,100)], np.linspace(0.0,0.0035,100), '-',
        label='average doc length')
ax.legend()
ax.grid()
fig.tight_layout()
plt.show()


# In[12]:


#%% only keep articles with more than 20 tokens, otherwise too short
# only keep articles with more than 30 tokens, otherwise too short
df = df[df['tokenized'].map(len) >= 30]
# make sure all tokenized items are lists
df = df[df['tokenized'].map(type) == list]
df.reset_index(drop=True,inplace=True)
print("After cleaning and excluding short aticles, the dataframe now has:", len(df), "articles")
df.head()


# In[13]:


lda=models.LdaModel.load('lda.model')
dictionary= corpora.Dictionary.load('dictionary.dict')
```

```

# In[45]:


#%%What we testing against
train_set = df[df.Type == 'rec.motorcycles']

corpus_motor_train = [dictionary.doc2bow(doc) for doc in train_set['tokenized']]


# In[46]:


#%%SIMILARITIES greate data stuff

newsgroups_test = fetch_20newsgroups(subset='test', remove=('headers', 'footers', 'quotes'))
sections = list(newsgroups_train.target_names)


df_test = pd.DataFrame(columns = ['Type', 'Data'])

data = []
names = []
for i in range(len(sections)):
    d = fetch_20newsgroups(shuffle=True, random_state=1,subset = 'test', remove=('headers', 'footers', 'quotes'),
    data.append(d.data)
    n = [sections[i]]*len(d.data)
    names.append(n)


# In[47]:



together_data = [item for sublist in data for item in sublist]
together_name = [item for sublist in names for item in sublist]

df_test = df_test.assign(Type = together_name, Data = together_data)

df_test['tokenized'] = df_test['Data'].apply(apply_all)


# In[49]:


#%% choose k and visually inspect the bottom 10 words of the top k
k = 17000
top_k_words_test = fdist_test.most_common(k)
```

```
top_k_words_test[-10:]
def keep_top_k_words(text):
    return [word for word in text if word in top_k_words_test]


# In[50]:


df['tokenized'] = df['tokenized'].apply(keep_top_k_words)


# In[51]:


#%% define a function only to keep words in the top k words
top_k_words_test,_ = zip(*fdist_test.most_common(k))
top_k_words_test = set(top_k_words_test)
df_test['tokenized'] = df_test['tokenized'].apply(keep_top_k_words)


# In[52]:


#%% document length
df_test['doc_len'] = df_test['tokenized'].apply(lambda x: len(x))
doc_lengths_test = list(df_test['doc_len'])
df_test.drop(labels='doc_len', axis=1, inplace=True)


# In[53]:


#%% only keep articles with more than 20 tokens, otherwise too short
df_test = df_test[df_test['tokenized'].map(len) >= 30]


# In[54]:


# make sure all tokenized items are lists
df_test = df_test[df_test['tokenized'].map(type) == list]
df_test.reset_index(drop=True,inplace=True)
print("After cleaning and excluding short aticles, the dataframe now has:", len(df_test), "articles")
df_test.head()


# In[55]:

```

```
#%%test motor
tm = df_test[df_test.Type == 'rec.motorcycles']


# In[56]:


#%%
dictionary_tm = corpora.Dictionary(tm['tokenized'])
new_bow = [dictionary.doc2bow(doc) for doc in tm['tokenized']]
new_doc_distribution_tm = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])


# In[58]:


#%% we need to use nested list comprehension here
# this may take 1-2 minutes...
doc_topic_dist = np.array([[tup[1] for tup in lst] for lst in lda[corpus_motor_train]])
doc_topic_dist.shape


# In[59]:


#%% this is surprisingly fast
#most_sim_ids = get_most_similar_documents(new_doc_distribution,doc_topic_dist)
all_sims_tm = []
for i in range(len(new_doc_distribution_tm)):
    doc_sims = jensen_shannon(new_doc_distribution_tm[i],doc_topic_dist)
    all_sims_tm.append(doc_sims)
#most_similar_df['title']

tm = [item for sublist in all_sims_tm for item in sublist]


# In[61]:


#%%SIMILARITIES TP

tp = df_test[df_test.Type == 'rec.autos']


# In[62]:
```

```
#%%
dictionary_tp = corpora.Dictionary(tp['tokenized'])
new_bow = [dictionary.doc2bow(doc) for doc in tp['tokenized']]
new_doc_distribution_tp = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])


# In[63]:


#%% this is surprisingly fast
#most_sim_ids = get_most_similar_documents(new_doc_distribution,doc_topic_dist)
all_sims_tp = []
for i in range(len(new_doc_distribution_tp)):
    doc_sims = jensen_shannon(new_doc_distribution_tp[i],doc_topic_dist)
    all_sims_tp.append(doc_sims)
#most_similar_df['title']


# In[64]:


tp = [item for sublist in all_sims_tp for item in sublist]


# In[66]:


#%% cdf
tm.sort()
tp.sort()
cdf_tm= 1. * np.arange(len(tm)) / (len(tm) - 1)
cdf_tp= 1. * np.arange(len(tp)) / (len(tp) - 1)
tm = np.array(tm)
tp = np.array(tp)

plt.scatter(tm,cdf_tm, s = 0.5, color = 'g', label = 'moto(train) v moto(test)')
plt.scatter(tp,cdf_tp, s = 0.5, color = 'r', label = 'moto(train) v auto(test)')

plt.legend()
plt.show


# In[67]:


from scipy import stats
```

```
# In[68]:


x = stats.ks_2samp(tm, tp)


# In[69]:


print(x)


# In[70]:


x.statistic


# In[71]:


x.pvalue
```

Motorcycle/ Politics Similarity Analysis Word2vec

```
#!/usr/bin/env python
# coding: utf-8
import pandas as pd
import numpy as np
import gensim
import time
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")
from sklearn.datasets import fetch_20newsgroups
from nltk.stem import WordNetLemmatizer
from gensim.models import Word2Vec,KeyedVectors
from gensim.test.utils import common_texts
from gensim.corpora import Dictionary
from gensim.models import Word2Vec, WordEmbeddingSimilarityIndex
from gensim.similarities import SoftCosineSimilarity, SparseTermSimilarityMatrix
import seaborn as sns
from gensim.corpora import Dictionary
import gensim
# upgrade gensim if you can't import softcossim
```

```python
from gensim.matutils import softcossim
from gensim import corpora
from gensim.utils import simple_preprocess


# In[2]:


#%% Download data
newsgroups_train = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'))
sections = list(newsgroups_train.target_names)


df = pd.DataFrame(columns = ['Type', 'Data'])

data = []
names = []
for i in range(len(sections)):
    d = fetch_20newsgroups(shuffle=True, random_state=1,subset = 'train', remove=('headers', 'footers', 'quotes')
    data.append(d.data)
    n = [sections[i]]*len(d.data)
    names.append(n)


# In[3]:


together_data = [item for sublist in data for item in sublist]
together_name = [item for sublist in names for item in sublist]

df = df.assign(Type = together_name, Data = together_data)


# In[4]:


len(df['Data'])


# In[5]:


#%%word2vec train function
def train_w2v(data):
    """
    This function trains the word2vec model
    """
    t1 = time.time()
```

```
    word2vec = Word2Vec(sentences=data, size=len(data), window=5, min_count=5, workers=4, sg=0)
    #index the words
    index2word_set = list(word2vec.wv.index2word)
    word_vectors = word2vec.wv #all information is stored in word vectors
    del word2vec #delete model to trim unneeded model state

    t2 = time.time()
    print("Time to train word2vec model on ", len(df), "articles: ", (t2-t1)/60, "min")
    return word_vectors, index2word_set

#%%cossim function
def create_soft_cossim_matrix(docsim_index, query_corpus, dictionary):
    all_soft_cossim = []
    for i in query_corpus:
        if len(i) > 0:
            sims = list(docsim_index[i])
            all_soft_cossim.extend(sims)
    return all_soft_cossim


# In[6]:


'''
#%% Apply model
word_vectors, index2word_set = train_w2v([simple_preprocess(doc) for doc in df['Data']])
#%%
termsim_index = WordEmbeddingSimilarityIndex(word_vectors)
#To compute soft cosines, you need the dictionary (a map of word to unique id),
#the corpus (word counts) for each sentence and the similarity matrix

# Prepare a dictionary and a corpus.
dictionary = corpora.Dictionary([simple_preprocess(doc) for doc in df['Data']])

# Prepare the similarity matrix
similarity_matrix = SparseTermSimilarityMatrix(termsim_index, dictionary)  # construct similarity matrix
'''


# In[7]:


word_vectors=KeyedVectors.load('word2vec.model')


# In[8]:


```

```
len(word_vectors.wv.vocab)


# In[9]:


dictionary = corpora.Dictionary.load('data_dictionary')


# In[10]:


similarity_matrix =SparseTermSimilarityMatrix.load('similarity_matrix')


# In[11]:


train_set = [dictionary.doc2bow(simple_preprocess(doc)) for doc in df[df.Type == 'rec.motorcycles']['Data']]
docsim_index = SoftCosineSimilarity(train_set, similarity_matrix)


# In[12]:


len(train_set)


# In[31]:


docsim_index[[(0,1),(6,2)]].shape


# In[32]:


#Now we are ready to calculate softcossim.
#What we testing against (Convert the train docs into bow)
#%%

#Get the test sets
df_test = pd.DataFrame(columns = ['Type', 'Data'])

data = []
names = []
for i in range(len(sections)):
    d = fetch_20newsgroups(shuffle=True, random_state=1,subset = 'test', remove=('headers', 'footers', 'quotes'),
```

```
    data.append(d.data)
    n = [sections[i]]*len(d.data)
    names.append(n)



# In[33]:


together_data = [item for sublist in data for item in sublist]
together_name = [item for sublist in names for item in sublist]

df_test = df_test.assign(Type = together_name, Data = together_data)
#Convert the test docs into bow
#bow_tm = [dictionary.doc2bow(simple_preprocess(doc)) for doc in df_test[df_test.Type == 'rec.motorcycles']['Data'
#bow_tp = [dictionary.doc2bow(simple_preprocess(doc)) for doc in df_test[df_test.Type == 'talk.politics.mideast'][
tm = [dictionary.doc2bow(simple_preprocess(doc)) for doc in df_test[df_test.Type == 'rec.motorcycles']['Data']]
tp = [dictionary.doc2bow(simple_preprocess(doc)) for doc in df_test[df_test.Type == 'talk.politics.mideast']['Dat


# In[34]:


len(tm)


# In[35]:


len(tp)


# In[11]:


all_sims_tm = create_soft_cossim_matrix(docsim_index, tm, dictionary)
#%%
all_sims_tp = create_soft_cossim_matrix(docsim_index, tp, dictionary)


# In[15]:


#%% Hist
sns.distplot(all_sims_tm,hist=True, kde=True,label = 'Training motorcycles against test motorcycles')

sns.distplot(all_sims_tp,hist=True, kde=True, label = 'motorcycles against politics')
plt.title("Word2Vec similarity histogram")
```

```
plt.legend()
plt.show


# In[13]:


#%%
sns.distplot(all_sims_tm,hist_kws=dict(cumulative=True),
             kde_kws=dict(cumulative=True),label = 'Training motorcycles against test motorcycles')
sns.distplot(all_sims_tp,hist_kws=dict(cumulative=True),
             kde_kws=dict(cumulative=True), label = 'Training motorcycles against politics')
plt.title("CDF similarity histogram")
plt.legend()
plt.show


# In[21]:


#%% cdf
all_sims_tm.sort()
all_sims_tp.sort()

cdf_tm= 1. * np.arange(len(all_sims_tm)) / (len(all_sims_tm) - 1)
cdf_tp= 1. * np.arange(len(all_sims_tp)) / (len(all_sims_tp) - 1)
all_sims_tm_array = np.array(all_sims_tm)
all_sims_tp_array = np.array(all_sims_tp)


plt.scatter(all_sims_tm_array,cdf_tm, s = 0.5, color = 'green', label = 'Motorcycles(train) vs. motorcycles(test)
plt.scatter(all_sims_tp_array,cdf_tp, s = 0.5, color = 'red', label = 'Motorcycles(train) vs. politics(test)')
plt.title("Word2Vec similarity curve")


plt.legend()
plt.show


# In[16]:


from scipy import stats


# In[17]:


```

```python
x = stats.ks_2samp(cdf_tm, cdf_tp)


# In[18]:


print(x)


# In[19]:


x.statistic


# In[20]:


x.pvalue
```

LDA Performance for different K values

```python
#!/usr/bin/env python
# coding: utf-8

# # Fake News LDA Notebook

# In this notebook we explore the performance of the LDA on the task of relevance judgement. For this taks we use

# In[298]:


#import the required libraries for the experiment
import pandas as pd
import numpy as np
import nltk
from nltk.corpus import stopwords
import gensim
from gensim.models import LdaModel
from gensim import models, corpora, similarities
import re
from nltk.stem.porter import PorterStemmer
import time
from nltk import FreqDist
from scipy.stats import entropy
```

```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import seaborn as sns
sns.set_style("darkgrid")
from sklearn.datasets import fetch_20newsgroups
from nltk.stem import WordNetLemmatizer


# In[ ]:


#Import the fake news data
fake_data = pd.read_csv('fake.csv')


# In[ ]:


fake_data.head()


# In[ ]:


fake_data.tail()


# In[299]:


#Choose columns to train the model on the correct Fake news data set found at httest_bss://www.kaggle.com/mrisdal/
fake_data=fake_data[['text','type']]


# In[300]:


#Dimensions of data
print(fake_data.shape)


# ### Function definition

# In[301]:


def initial_clean(text):
    """
```

```python
    Function to clean text of websites, email addresess and any punctuation
    We also lower case the text
    """
    text = re.sub('\s+https://www.kaggle.com/mrisdal/fake-news', ' ', str(text))
    text = re.sub("((\S+)?(http(s)?)(\S+))|((\S+)?(www)(\S+))|((\S+)?(\@)(\S+)?)", " ", text)
    text = re.sub("[^a-zA-Z ]", "", text)
    text = text.lower() # lower case the text
    text = nltk.word_tokenize(text)
    return text

#import stopwords from nltk
stop_words = stopwords.words('english')
def remove_stop_words(text):
    """
    Function that removes all stopwords from text
    """
    return [word for word in text if word not in stop_words]

#Create lemmmayizer instance from WordNet
lmtzr = WordNetLemmatizer()
def stem_words(text):
    """
    Function to stem words, so plural and singular are treated the same
    """

    text = [lmtzr.lemmatize(word) for word in text]
    text = [word for word in text if len(word) > 1] # make sure we have no 1 letter words
    return text

def apply_all(text):
    """
    This function applies all the functions above into one
    """
    return stem_words(remove_stop_words(initial_clean(text)))

def jensen_shannon(query, matrix):
    """
    This function implements a Jensen-Shannon similarity
    between the input query (an LDA topic distribution for a document)
    and the entire corpus of topic distributions.
    It returns an array of length M where M is the number of documents in the corpus
    """
    # lets keep with the p,q notation above
    p = query[None,:].T # take transpose
    q = matrix.T # transpose matrix
    m = 0.5*(p + q)
    return np.sqrt(0.5*(entropy(p,m) + entropy(q,m)))
```

```
def get_most_similar_documents(query,matrix,k=10):
    """
    This function implements the Jensen-Shannon distance above
    and retruns the top k indices of the smallest jensen shannon distances
    """
    sims = jensen_shannon(query,matrix) # list of jensen shannon distances
    return sims.argsort()[:k] # the top k positional index of the smallest Jensen Shannon distances

def keep_top_k_words(text):
    return [word for word in text if word in top_k_words]




# In[302]:


fake_data['tokenized'] = fake_data['text'].apply(apply_all)


# In[303]:


#%%
# first get a list of all words
all_words = [word for item in list(fake_data['tokenized']) for word in item]
# use nltk fdist to get a frequency distribution of all words
fdist = FreqDist(all_words)


# In[304]:


#%% choose k and visually inspect the bottom 10 words of the top k
k = 17000
top_k_words = fdist.most_common(k)
top_k_words[-10:]


# In[305]:


#%% define a function only to keep words in the top k words
top_k_words,_ = zip(*fdist.most_common(k))
top_k_words = set(top_k_words)


# In[306]:
```

```


#%% document length
fake_data['doc_len'] = fake_data['tokenized'].apply(lambda x: len(x))
doc_lengths = list(fake_data['doc_len'])
fake_data.drop(labels='doc_len', axis=1, inplace=True)


# In[307]:


print("length of list:",len(doc_lengths),
      "\naverage document length", np.average(doc_lengths),
      "\nminimum document length", min(doc_lengths),
      "\nmaximum document length", max(doc_lengths))


# In[308]:


# plot a histogram of document length
num_bins = 1000
fig, ax = plt.subplots(figsize=(12,6));
# the histogram of the data
n, bins, patches = ax.hist(doc_lengths, num_bins, normed=1)
ax.set_xlabel('Document Length (tokens)', fontsize=15)
ax.set_ylabel('Normed Frequency', fontsize=15)
ax.grid()
ax.set_xticks(np.logspace(start=np.log10(50),stop=np.log10(2000),num=8, base=10.0))
plt.xlim(0,2000)
ax.plot([np.average(doc_lengths) for i in np.linspace(0.0,0.0035,100)], np.linspace(0.0,0.0035,100), '-',
        label='average doc length')
ax.legend()
ax.grid()
fig.tight_layout()
plt.show()


# ## Filtering Data for non empty documents

# In[309]:


#%% only keep articles with more than 20 tokens, otherwise too short
# only keep articles with more than 30 tokens, otherwise too short
fake_data = fake_data[fake_data['tokenized'].map(len) >= 30]
# make sure all tokenized items are lists
fake_data = fake_data[fake_data['tokenized'].map(type) == list]
```

```
fake_data.reset_index(drop=True,inplace=True)
print("After cleaning and excluding short aticles, the dataframe now has:", len(fake_data), "articles")
fake_data.head()


# ## Manual Train and test split
# We retain 90% if the 11349 documents fro training and the 10% is to test the performance of each model

# In[310]:


msk = np.random.rand(len(fake_data)) < 0.9
train_fake_data = fake_data[msk]
train_fake_data.reset_index(drop=True,inplace=True)

test_fake_data = fake_data[~msk]
test_fake_data.reset_index(drop=True,inplace=True)


# We have that there are 10201 documents for training

# In[311]:


train_fake_data.shape


# We seth the seed to 4 to get consisent results for the different topic models

# In[312]:


np.random.seed(4)


# In[313]:


def train_lda(data,corpus,dictionary,K):
    """
    This function trains the lda model
    We setup parameters like number of topics, the chunksize to use in Hoffman method
    We also do 2 passes of the data since this is a small dataset, so we want the distributions to stabilize
    """
    num_topics = K
    chunksize = 300
    t1 = time.time()
    # low alpha means each document is only represented by a small number of topics, and vice versa
```

```
    # low eta means each topic is only represented by a small number of words, and vice versa
    lda = LdaModel(corpus=corpus, num_topics=num_topics, id2word=dictionary,
                   alpha=1e-2, eta=0.5e-2, chunksize=chunksize, minimum_probability=0.0, passes=2)
    t2 = time.time()
    print("Time to train LDA model on ", len(data), "articles: ", (t2-t1)/60, "min")
    return lda


# Topic Sizes we will consider

# In[314]:


num_of_topics=[50,100,200,250,300]


# In[315]:


#The dictionaryand corpus are kept constant
fake_dictionary = corpora.Dictionary(train_fake_data['tokenized'])
fake_corpus = [fake_dictionary.doc2bow(doc) for doc in train_fake_data['tokenized']]


# In[316]:


#%%LDA train function
#%% Apply model for different topic sizes
for k in num_of_topics:
    lda = train_lda(train_fake_data,fake_corpus,fake_dictionary,k)
    lda.save('lda_fakenews'+str(k)+'.model')



# In[317]:


#fake_dictionary.save('fakenews_dictionary')


# In[318]:


#dictionary = corpora.Dictionary.load('fakenews_dictionary')


# In[319]:
```

```
#lda=LdaModel.load('lda_fakenews.model')


# In[320]:


#Number of documents in our test set


# In[321]:


test_fake_data.shape


# In[322]:


#%%text documents for t``````ype bias
test_bias = test_fake_data[test_fake_data.type == 'bias']


# In[323]:


test_bias.head(20)


# Topic K=50,100,200,250,300

# In[324]:


lda=LdaModel.load('lda_fakenews250.model')


# In[325]:


dictionary_test_bias = corpora.Dictionary(test_bias['tokenized'])
new_bow = [dictionary_test_bias.doc2bow(doc) for doc in test_bias['tokenized']]
new_doc_distribution_test_bias = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])


# In[326]:

```

```
test_bs = test_fake_data[test_fake_data.type == 'bs']


# In[327]:


#%%
dictionary_test_bs = corpora.Dictionary(test_bs['tokenized'])
new_bow = [dictionary_test_bs.doc2bow(doc) for doc in test_bs['tokenized']]
new_doc_distribution_test_bs = np.array([[tup[1] for tup in lst] for lst in lda[new_bow]])


# In[328]:


doc_topic_dist = np.array([[tup[1] for tup in lst] for lst in lda[fake_corpus]])
doc_topic_dist.shape


# In[329]:


all_sims_test_bias = []
doc_sims_bias=[]
for i in range(len(new_doc_distribution_test_bias)):
    doc_sims = jensen_shannon(new_doc_distribution_test_bias[i],doc_topic_dist)
    all_sims_test_bias.append(1-np.mean(doc_sims))
    doc_sims_bias.append(doc_sims)

all_sims_test_bias=[x for x in all_sims_test_bias if str(x) != 'nan']
#test_bias = [item for sublist in all_sims_test_bias for item in sublist]


# In[330]:


#%% this is surprisingly fast
#most_sim_ids = get_most_similar_documents(new_doc_distribution,doc_topic_dist)
doc_sim_test_bs=[]
all_sims_test_bs = []
for i in range(len(new_doc_distribution_test_bs)):
    doc_sims = jensen_shannon(new_doc_distribution_test_bs[i],doc_topic_dist)
    all_sims_test_bs.append(1-np.mean(doc_sims))
    doc_sim_test_bs.append(doc_sims)

all_sims_test_bs=[x for x in all_sims_test_bs if str(x) != 'nan']
#test_bs = [item for sublist in all_sims_test_bs for item in sublist]
```

```python


# In[331]:


#%% cdf
all_sims_test_bias.sort()
all_sims_test_bs.sort()
cdf_tm= 1. * np.arange(len(all_sims_test_bias)) / (len(all_sims_test_bias) - 1)
cdf_tp= 1. * np.arange(len(all_sims_test_bs)) / (len(all_sims_test_bs) - 1)
all_sims_test_bias = np.array(all_sims_test_bias)
all_sims_test_bs = np.array(all_sims_test_bs)

plt.scatter(all_sims_test_bias,cdf_tm, s = 0.9, color = 'g', label = 'Bias')
plt.scatter(all_sims_test_bs,cdf_tp, s = 0.9,color = 'r', label = 'Bs')

plt.legend()
plt.show


# In[332]:



all_sims_test_bs.shape


# In[333]:


all_sims=list(all_sims_test_bias)+list(all_sims_test_bs)


# In[334]:


test_bias_label=['b']*len(all_sims_test_bias)
test_bs_label=['r']*len(all_sims_test_bs)


# In[335]:


labels=test_bias_label+test_bs_label


# In[336]:

```

```
len(all_sims)


# In[337]:


# create dataframe of rel_index and label
df_original = pd.DataFrame({'label': labels,
                                'rel_index': all_sims})


# In[338]:


df_original.head()


# In[339]:


df=df_original.sort_values('rel_index',ascending=False)


# In[340]:


# find threshold (minimum relevance index of relevant articles)
threshold = df.loc[df['label'] == 'b']['rel_index'].min()

# true relevant (relevant articles above threshold)
true_relevant = df.loc[(df["label"] == 'b') & (df["rel_index"] > threshold)]
false_relevant = df.loc[(df["label"] == 'b') & (df["rel_index"] < threshold)]

# true irrelevant (irrelevant articles below threshold)
true_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] < threshold)]
false_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] > threshold)]

perc_ignore = float(len(true_irrelevant)) / (len(df)) * 100

accuracy=(len(true_relevant)+len(true_irrelevant))/ len(df)
try:
    recall=float(len(true_irrelevant)) / float(len(false_relevant)+len(true_irrelevant))
except ZeroDivisionError:
    recall =0
    # false irrelevant (irrelevant articles above threshold)
false_irrelevant = df.loc[(df["label"] == 'r') & (df["rel_index"] >= threshold)]
```

```python
plt.figure(figsize=(25, 10))
plt.ylim([-1,max(df['rel_index']+1)])
plt.xlim([0, len(labels)])
plt.scatter(range(len(df)), df['rel_index'], c=df['label'], s=30,alpha=0.7)
# plot threshold
plt.axhline(threshold, c='black', linewidth=1.5)
plt.xlabel('Number of Documents',fontsize=20)
plt.ylabel('Relevance index',fontsize=20)
plt.title('Relevance Jugdement for LDA K=250',fontsize=20)
plt.text(500, 1.0, r'True irrelevant: ' + str(len(true_irrelevant)) + '\n' + 'From total of: ' + str(
        len(df)) + '  (' + "%.2f" % perc_ignore + '%)', verticalalignment='bottom', horizontalalignment='left')


# In[341]:


from scipy import stats


# In[342]:


x = stats.ks_2samp(all_sims_test_bias, all_sims_test_bs)


# In[ ]:


print(x)


# In[ ]:


x.statistic


# In[ ]:


x.pvalue


# # Coherence

# In[ ]:


```

```
from gensim.models.coherencemodel import CoherenceModel


# In[ ]:


m1=LdaModel.load('lda_fakenews50.model')
m2=LdaModel.load('lda_fakenews100.model')
m3=LdaModel.load('lda_fakenews200.model')
m4=LdaModel.load('lda_fakenews250.model')
m5=LdaModel.load('lda_fakenews300.model')


# In[ ]:



cm1 = CoherenceModel.for_models([m1, m2], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm1.get_coherence()


# In[ ]:


cm2 = CoherenceModel.for_models([m1, m3], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm2.get_coherence()


# In[ ]:


cm3 = CoherenceModel.for_models([m1, m3], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm3.get_coherence()


# In[ ]:


cm4 = CoherenceModel.for_models([m1, m4], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm4.get_coherence()


# In[ ]:


cm5 = CoherenceModel.for_models([m1, m5], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm5.get_coherence()

```

```python
# In[ ]:


cm23 = CoherenceModel.for_models([m2, m3], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm23.get_coherence()


# In[ ]:


cm24 = CoherenceModel.for_models([m2, m4], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm24.get_coherence()


# In[ ]:


cm25 = CoherenceModel.for_models([m2, m5], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm25.get_coherence()


# In[ ]:


cm34 = CoherenceModel.for_models([m3, m4], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm34.get_coherence()


# In[ ]:


cm35 = CoherenceModel.for_models([m3, m5], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm35.get_coherence()


# In[ ]:


cm45 = CoherenceModel.for_models([m4, m5], fake_dictionary, corpus=fake_corpus, coherence='u_mass')
cm45.get_coherence()
```