1. Introduction

Natural language processing (NLP) is all about creating systems that process or "understand" language in order to perform certain tasks. Once you have identified, extracted, and cleansed the content needed for your use case, the next step is to have an understanding of that content. In many use cases, the content with the most important information is written down in a natural language (such as German, English, Spanish, Chinese, etc.) and not conveniently tagged. To extract information from this content you will need to rely on some levels of text mining, text extraction, or possibly full-up natural language processing (NLP) techniques.
Typical full-text extraction for Internet content includes:

Extracting entities – such as companies, people, dollar amounts, key initiatives, etc.

Categorizing content – positive or negative (e.g. sentiment analysis), by function, intention or purpose, or by industry or other categories for analytics and trending

Clustering content – to identify main topics of discourse and/or to discover new topics

Fact extraction – to fill databases with structured information for analysis, visualization, trending, or alerts

Relationship extraction – to fill out graph databases to explore real-world relationships

1.1 Language Modeling

A statistical **language model** is a probability distribution over sequences of words. For example if we have two sentences have the same acoustic signal. We must have some intrinsic preference over sentences. Language modeling tries to capture the notion that some sentences are more likely than others by density estimation**.**

There are basically two types of language Modeling

- Unigram Language Model: A unigram language model makes the strong independence assumption that words are generated independently from a multinomial distribution.

- N-gram Language Models: In this kind of modeling one can partially incorporate such order into a language model by making weaker independence assumptions.

1.2 Spelling Correction

Spelling Correction uses a statistical model to find the original word, and returns a confidence value for each of them. In this algorithm we generally, identify and correct spelling mistakes in given string.

Example:

Parameter 1: word of string

```
{
 "word": "bananna"
}
```

Output:

```
[
  {
   "word": "banana",
   "confidence": 1.0
  }
]
```

1.3 POS tagging

Automatic assignment of descriptors to the given tokens is called Tagging. The descriptor is called tag. The tag may indicate one of the parts-of-speech, semantic information, and so on. So tagging a kind of classification.

A Part-Of-Speech Tagger (POS Tagger) is a piece of software that reads text in some language and assigns parts of speech to each word (and other token), such as nouns, verbs, adverbs, adjectives, pronouns, conjunction and their sub-categories..etc although generally computational applications use more fine-grained POS tags like 'noun-plural'.

Example

Word: Paper, Tag: Noun
Word: Go, Tag:Verb
Word: Beautiful, Tag:Adjective

ARCHITECTURE OF POS TAGGER

i. Tokenization: The given text is divided into tokens so that they can be used for further analysis. The tokens may be words, punctuation marks, and utterance boundaries.

ii. Ambiguity look-up: This is to use lexicon and a guessor for unknown words. While lexicon provides list of word forms and their likely parts of speech, guessors analyze unknown tokens. Compiler or interpreter, lexicon and guessor make what is known as lexical analyzer.

Iii. Ambiguity Resolution: This is also called disambiguation. Disambiguation is based on information about word such as the probability of the word. For example, power is more likely used as noun than as verb.

1.4 Models for Sequential tagging – MaxEnt

Maximum entropy modeling is a framework for integrating information from many heterogeneous information sources for classification. The data for a classification problem is described as a (potentially large) number of features. These features can be quite complex and allow the experimenter to make use of prior knowledge about what types of informations are expected to be important for classification. Each feature corresponds to a constraint on the model. We then compute the maximum entropy model, the model with the maximum entropy of all the models that satisfy the constraints. If we chose a model with less entropy, we would add `information' constraints to the model that are not justified by the empirical evidence available to us. We introduce the concept of maximum entropy through a simple example. Suppose we wish to model an expert translator's decisions concerning the proper French rendering of the English word in. Our model p of the expert's decisions assigns to each French word or phrase f an estimate, p(f), of the probability that the expert would choose f as a translation of in. To guide us in developing p, we collect a large sample of instances of the expert's decisions. Our goal is to extract a set of facts about the decision-making process from the sample (the first task of modeling) that will aid us in constructing a model of this process (the second task).

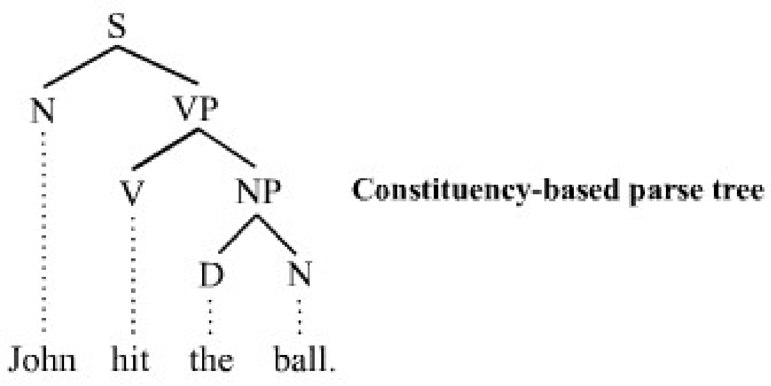
1.5 Models for Sequential tagging – CRF

It is Probabilistic Models for Segmenting and Labeling the Sequence Data. CRF has a single exponential model for the joint probability of the entire sequence of labels given the observation sequence. **Conditional random fields (CRFs)** are a class of statistical modeling method often applied in pattern recognition and machine learning and used for structured prediction. CRFs fall into the sequence modeling family. Whereas a discrete classifier predicts a label for a single sample without considering "neighboring" samples, a CRF can take context into account; e.g., the linear chain CRF (which is popular in natural language processing) predicts sequences of labels for sequences of input samples.

CRFs are a type of discriminative undirected probabilistic graphical model. It is used to encode known relationships between observations and construct consistent interpretations. It is often used for labeling or parsing of sequential data, such as natural language processing or biological sequences and in computer vision. Specifically, CRFs find applications in POS Tagging, shallow parsing, named entity recognition, gene finding and peptide critical functional region finding, among other tasks, being an alternative to the related hidden Markov models (HMMs). In computer vision, CRFs are often used for object recognition and image segmentation.

1.6 Syntax – Constituency Parsing

The constituency-based parse trees of constituency grammars distinguish between terminal and non-terminal nodes. The interior nodes are labeled by non-terminal categories of the grammar, while the leaf nodes are labeled by terminal categories. For example we get as



The parse tree is the entire structure, starting from S and ending in each of the leaf nodes (*John*, *hit*, *the*, *ball*). The following abbreviations are used in the tree:

S for sentence, the top-level structure in this example

NP for noun phrase. The first (leftmost) NP, a single noun "John", serves as the subject of the sentence. The second one is the object of the sentence.

VP for verb phrase, which serves as the predicate

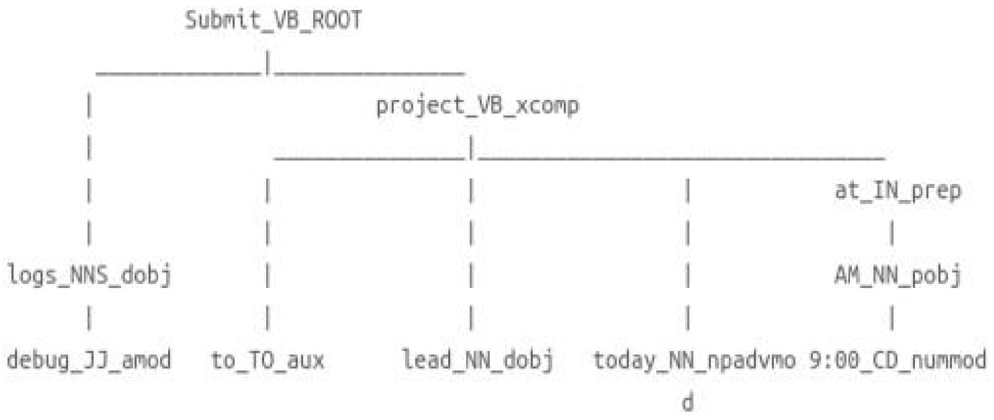
V for verb. In this case, it's a transitive verb *hit*.

D for determiner, in this instance the definite article "the"

N for noun

1.7 Dependency Parsing

Syntactic Parsing or Dependency Parsing is the task of recognizing a sentence and assigning a syntactic structure to it. The most widely used syntactic structure is the parse tree which can be generated using some parsing algorithms. These parse trees are useful in various applications like grammar checking or more importantly it plays a critical role in the semantic analysis stage. For example



```
              Submit_VB_ROOT
     ____________|_____________
    |                     project_VB_xcomp
    |            ______________|_____________________________
    |           |             |              |          at_IN_prep
    |           |             |              |              |
logs_NNS_dobj   |             |              |          AM_NN_pobj
    |           |             |              |              |
debug_JJ_amod  to_TO_aux  lead_NN_dobj  today_NN_npadvmo 9:00_CD_nummod
                                             d
```

1.8 Lexical Semantics

The lexical semantic analysis (LxSA) for the problem of detecting lexical units in text and assigning semantic information to these units.

Lexical semantics fits into linguistics curricula in various ways. Some of the most common ways are:

i) As a sub-module in a semantics course (often lower-mid level in degree)

ii) As part of a course on vocabulary/lexicology—including morphology, etymology, lexicography iii) As well as semantics (often lower-mid level)

iii) As a free-standing course (often upper level)

Lexical semantics courses can incorporate other disciplinary interests. Some lexical semantic topics are as below:

Pragmatics – No semantics course can help but to tread on the toes of pragmatics, and some theoretical approaches (particularly Cognitive Linguistics) have done away with the distinction between semantic and pragmatic competence.

Morphology – Just as there are many interfaces between syntax and sentential semantics, so there are between morphology and lexical semantics.

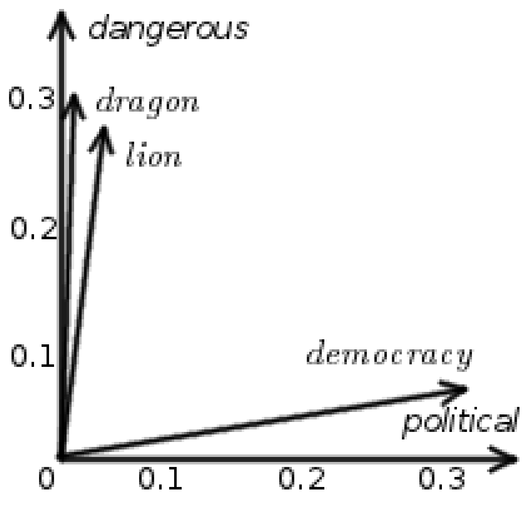
Psycholinguistics – Most lexical semantic issues can be addressed from a psycholinguistic perspective, and psycholinguistic methods offer evidence concerning how words and meanings are organised in the mind.

Language acquisition – Unlike grammar, vocabulary is acquired throughout life, so some of the issues in lexical acquisition can be addressed from an adult first- or second-language angle.

Computational linguistics – Much lexical semantic work nowadays is done in computational linguistics/natural language processing (NLP), including much work on polysemy/ambiguity resolution and the development of semantic networks.

1.9 Distributional semantics

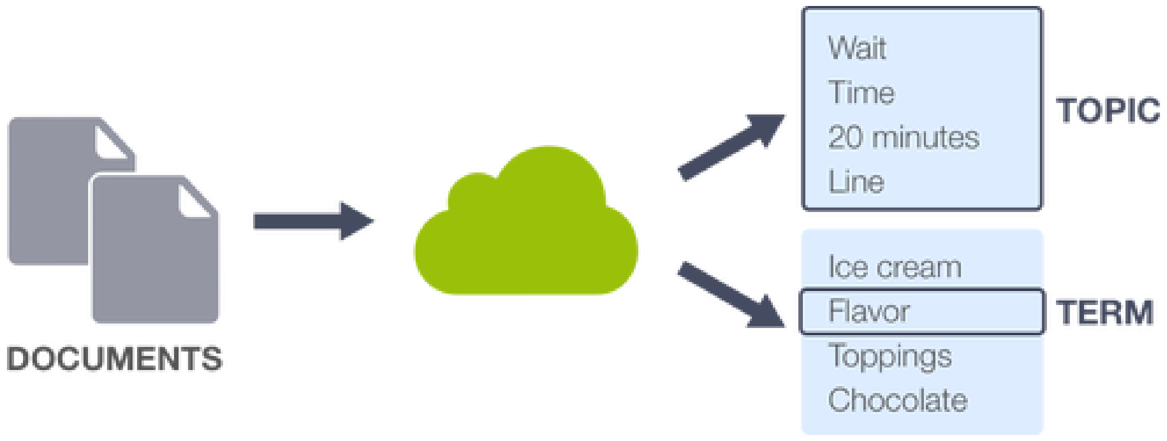
Distributional semantics is a subfield of Natural Language Processing that learns meaning from word usages. If we say that meaning comes from use, we can derive a model of meaning from observable uses of language. This is what distributional semantics does, in a somewhat constrained fashion. A typical way to get an approximation of the meaning of words in distributional semantics is to look at their linguistic context in a very large corpus. Here is a very simplified example, where I define the words ‘dragon’, ‘lion’ and ‘democracy’ with respect to only two dimensions: ‘dangerous’ and ‘political’.



2. Topic Model

There are plenty of valuable insights hidden in your text data. Plain text data can be very useful for content recommendation, information retrieval tasks, segmenting your data, or training predictive models. The standard “bag of words” analysis BigML performs when it creates your dataset is often useful, but sometimes it doesn’t go far enough as there may be hidden patterns in text data that are difficult to discover when you’re only considering occurrences of a single word at a time. Often, the **Latent Dirichlet Allocation(LDA)** algorithm is able to organize your text data in such a way that it causes some of this hidden information to spring to the fore.

There are three key vocabulary words we need to know when we’re trying to understand **the basics of Topic Models: documents, terms, and topics**. Latent Dirichlet Allocation (LDA) is an unsupervised learning method that discovers different topics underlying a collection of documents, where each document is a collection of words, or *terms*. LDA assumes that any document is a combination of one or more topics, and each topic is associated with certain high probability terms.



In natural language processing, a probabilistic topic model describes the semantic structure of a collection of documents, the so-called corpus. Latent Dirichlet allocation (LDA) is one of the most popular and successful models to discover common topics as a hidden structure of the collection of documents. According to the LDA model, text documents are represented by mixtures of topics. This means that a document concerns one or multiple topics in different proportions. A topic can be viewed as a cluster of similar words. More formally, the model assumes each topic to be characterized by a distribution over a fixed vocabulary, and each text document to be generated by a distribution of topics.
The basic assumption of LDA is that the documents have been generated in a two-step random process rather than having been written by a human. The generative process for a document consisting of *N* words is as follows. The most important model parameter is the number of topics *k* that has to be chosen in advance. In the first step, the mixture of topics is generated according to a Dirichlet distribution of *k*topics. Second, from the previously determined topic distribution, a topic is randomly chosen, which then generates a word from its distribution over the vocabulary. The second step is repeated for the *N* words of the document. Note that LDA is a bag-of-words model and the order of words appearing in the text as well as the order of the documents in the collection is neglected.

Data Preprocessing

We follow a typical workflow of data preparation for natural language processing (NLP). Textual data is transformed into numerical feature vectors required as input for the LDA machine learning algorithm.

The textual data extracted from the HTML file is then normalized by removing numbers, punctuation and other special characters and using lowercase. A so-called tokenizer splits the sentences into words (tokens) that are separated by whitespace.

```
extractText = udf(
 lambda d: BeautifulSoup(d, "lxml").get_text(strip=False), StringType())
removePunct = udf(
 lambda s: re.sub(r'[^a-zA-Z0-9]', r' ', s).strip().lower(), StringType())

# normalize the post content (remove html tags, punctuation and lower case..)
df_posts_norm = df_posts.withColumn("text", removePunct(extractText(df_posts.post_content)))

# breaking text into words
tokenizer = RegexTokenizer(inputCol="text", outputCol="words",
                           gaps=True, pattern=r'\s+', minTokenLength=2)
df_tokens = tokenizer.transform(df_posts_norm)
```

Language identification

We only want to analyze English blog posts and have to identify the language since no such tag is available in our data set. A simple classification between English and German as the primary language is achieved by comparing the fraction of stop words in the text. The Fraction of English stop words in a given article is obtained by counting the number of English stop words that appear at least once in the text, divided by the total number of stop words in the list. Similarly, we calculate the fraction of German stop words and decide which language an article mainly uses by the larger of the two fractions.

```
from nltk.corpus import stopwords
englishSW = set(stopwords.words('english'))
germanSW = set(stopwords.words('german'))

nEngSW = len(englishSW)
nGerSW = len(germanSW)

RatioEng = udf(lambda l: len(set(l).intersection(englishSW)) / nEngSW)
RatioGer = udf(lambda l: len(set(l).intersection(germanSW)) / nGerSW)

df_tokens_en = (df_tokens.withColumn("ratio_en", RatioEng(df_tokens['words']))
                .withColumn("ratio_ge", RatioGer(df_tokens['words']))
                .withColumn("Eng", col('ratio_en') > col('ratio_ge'))
                .filter('Eng'))
```

Filtering out stop words and stemming

The last preprocessing steps are filtering out the English stop words, as these common words presumably do not help in identifying meaningful topics, and stemming the words such that, for instance, “test”, “tests”, “tested”, and “testing” are all reduced to their word stem “test”. After having trained an LDA model, we inspect the topics and identify additional stop words, which are filtered out for the subsequent model training. This procedure is repeated, as long as stop words appear in the lists of top words.

```
df_finalTokens = swRemover.transform(df_tokens_en)

# Stemming
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer("english", ignore_stopwords=False)
udfStemmer = udf(lambda l: [stemmer.stem(s) for s in l], ArrayType(StringType()))

df_finalTokens = df_finalTokens.withColumn("filteredStemmed",
                                         udfStemmer(df_finalTokens["filtered"]))
```

Feature generation

Each document is represented as a vector of counts, the length of which is given by the number of words in the vocabulary. The CountVectorizer is an estimator that generates a model from which the tokenized documents are transformed into count vectors. Words have to appear at least in two different documents and at least four times in a document to be taken into account.

```
cv = CountVectorizer(inputCol="filteredStemmed", outputCol="features", vocabSize=2500,
minDF=2, minTF=4)

cvModel = cv.fit(df_finalTokens)

countVectors = (cvModel
               .transform(df_finalTokens)
               .select("ID", "features").cache())

cvModel.save("path/to/model/file")
```

Model Training and Evaluation

Data is incrementally processed in small batches, which allows scaling to very large data sets that might even arrive in a streaming fashion.

```
df_training, df_testing = countVectors.randomSplit([0.9, 0.1], 1)

numTopics = 20 # number of topics
```

```
lda = LDA(k = numTopics, seed = 1, optimizer="online", optimizeDocConcentration=True,
 maxIter = 50,           # number of iterations
 learningDecay = 0.51,   # kappa, learning rate
 learningOffset = 64.0,  # tau_0, larger values downweigh early iterations
 subsamplingRate = 0.05, # mini batch fraction
 )

ldaModel = lda.fit(df_training)

lperplexity = ldaModel.logPerplexity(df_testing)

ldaModel.save(path)
```

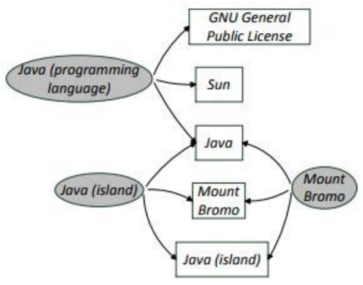
In general, the data set is split into a training set and a testing set in order to evaluate the model performance via a measure such as the perplexity, i.e., a measure of how well the word counts of the test documents are represented by the topic's word distributions. However, we find it more useful to evaluate the model manually by looking at the resulting topics and the corresponding distribution of words.

3. Entity Linking, Information Extraction

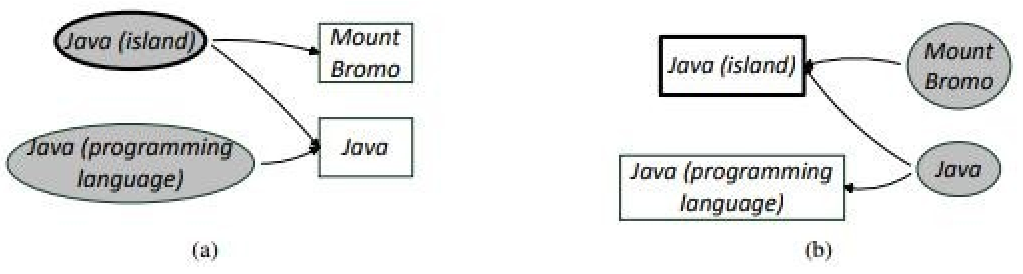
Most of them are abbreviations and Internet slang expressions with different meanings. Having a dictionary encompassing as many as entities and their specific items is a firm basis in entity recognition and therefore important. Initially different items of an entity are obtained by searching Baidu Encyclopedia web pages, we then sort out different items of an entity according to their occurrence in descending order.

We segment the context document into word or name phrase fragments and filter out stop words (e.g. about, have, the, etc.). In order to evaluate our entity linking method in different scales, we select nodes of neighboring entities from these fragments in several context window sizes around the target mention name: the sentence where the target name appears in, plus the immediately adjacent sentence before and after the sentence containing the target name, and plus the adjacent two sentences before and after, etc

Take an example of wikipedia graph as below:



An example of the graph-based named entity disambiguation method as below



4. Text Summarization

The Text Summarization identifies the most important points of a text and expresses them in a shorter document. Summarization process:

i) Interpret the text

ii) Extract the relevant information (topics of the source)

iii) Condense extracted information and create summary representation

iv) Present summary representation to reader in natural language

Text Summarization with Gensim

Text summarization is one of the newest and most exciting fields in NLP, allowing for developers to quickly find meaning and extract key words and phrases from documents. RaRe Technologies' newest intern, Ólavur Mortensen, walks the user through text summarization features in Gensim.

In the code below, we read the text file directly from a web-page using "requests". Then we produce a summary and some keywords.

```
import requests
from gensim.summarization import keywords
from gensim.summarization import summarize

text = requests.get('http://theeditorsblog.net/2012/07/15/clear-the-dread-from-the-dreaded-synopsis/').text

print('Summary:')
print(summarize(text, ratio=0.01))

print('\nKeywords:')
print(keywords(text, ratio=0.01))
```

Output will be:

```
Summary:
<p><strong>On one side is the idea that the synopsis is a tease, a draw to get the proper persons
(agents, editors, contest judges) to want to read the full story</strong>.
If you&#8217;ve ever judged a writing contest or entered one that makes use of a synopsis,
you&#8217;re probably used to this style.
<p>Those looking for this style in a synopsis aren&#8217;t looking to read a mini-version of the
novel; while they want to know what happens, they don&#8217;t need to see your writing style.
<p>With these true differences between the approaches, is it any wonder that writers feel confused
about what to include, what to leave out, and how to word the synopsis when <strong>one
recommendation is to write in the style of the novel and another is to simply convey, in the style of a
police report, what happens</strong>?
If you&#8217;re including a synopsis for a contest, you&#8217;ll need one that&#8217;s well
written and appealing, that reads like your story, that engages the reader.
I could probably easily tell the entire story in a page or two, but in order to include all the important
plot points, the main characters, their development, and the ending I&#8217;ve had to use up the
entire 10 pages.</p>
When including bits of dialogue or lines from the story in a synopsis to quickly &#8216;convey
flavor and emotion&#8217; &#8211; how do you distinguish those lines from those you are writing
as the observing third person?
When including bits of dialogue or lines from the story in a synopsis to quickly ‘convey flavor and
emotion’ – how do you distinguish those lines from those you are writing as the observing third
person?
<p>Imani, submit only the synopsis of the first book and don&#8217;t mention it&#8217;s part of a
series in your query letter unless you know beyond doubt that the agent or publisher is looking for a
series.
Keywords:
class
title
div
style
styles
http
synopsis
tag
tags
href
```

5. Text Classification using scikit-learn, python and NLTK

The Documents/Text Classification is one of the important and typical task in *supervise* machine learning (ML). Assigning categories to documents, which can be a web page, library book, media articles, gallery etc. has many applications like e.g. spam filtering, email routing, sentiment analysis etc.

Text Classification can be done by following steps as below

Environment setting and some Prerequisite

The prerequisites to follow this example are python version 2.7.3 and jupyter notebook.You can just install anaconda and it will get everything for you.Also, little bit of python and ML basics including text classification is required. We will be using scikit-learn (python) libraries for our example.

Loading the data set

The data set will be using for this example is the famous “20 Newsgoup” data set.

The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. To the best of my knowledge, it was originally collected by Ken Lang, probably for his Newsweeder: Learning to filter netnews paper, though he does not explicitly mention this collection. The 20 newsgroups collection has become a popular data set for experiments in text applications of machine learning techniques, such as text classification and text clustering.

```
from sklearn.datasets import fetch_20newsgroups
twenty_train = fetch_20newsgroups(subset='train', shuffle=True)


twenty_train.target_names #prints all the categories
print("\n".join(twenty_train.data[0].split("\n")[:3])) #prints first line of the first data file
```

Features extraction from text files

Text files are actually series of words (ordered). In order to run machine learning algorithms we need to convert the text files into numerical feature vectors.

```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()
X_train_counts = count_vect.fit_transform(twenty_train.data)
X_train_counts.shape
```

Here by doing ‘count_vect.fit_transform(twenty_train.data)’, we are learning the vocabulary dictionary and it returns a Document-Term matrix. [n_samples, n_features].

We use the following code for achieving Document-Term matrix

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidf_transformer = TfidfTransformer()
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
X_train_tfidf.shape
```

Running Machine Learning Algorithm

There are various algorithms which can be used for text classification. We will start with the most simplest one '**Naive Bayes'**(NB) algorithm.

```
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB().fit(X_train_tfidf, twenty_train.target)
```

6. Summarize Documents using Tf-Idf

A Term Frequency is a count of how many times a word occurs in a given document. The Inverse Document Frequency is the the number of times a word occurs in a corpus of documents. Tf-idf is used to weight words according to how important they are. Words that are used frequently in many documents will have a lower weighting while infrequent ones will have a higher weighting. Below is an explanation from Wikipedia.

The tf-idf value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.

The following terms are used to Summarize Documents

Preprocessing the document

There are a number of preprocessing techniques that can be applied to a document. The most obvious being the removal of non alpha-numeric characters, stop words, and any unnecessary punctuation. Below is an example of how to remove a few punctuation marks.

```
document = re.sub('[^A-Za-z .-]+', ' ', document)
document = document.replace('-', '')
document = document.replace('…', '')
document = document.replace('Mr.', 'Mr').replace('Mrs.', 'Mrs')
```

Creating a count vector

To create a count vector we'll need to implement sklearn's *CountVectorizer* and fit it with the corpus.

```
from sklearn.feature_extraction.text import CountVectorizer
count_vect = CountVectorizer()
count_vect = count_vect.fit(train_data)
freq_term_matrix = count_vect.transform(train_data)
```

Building the tf-idf matrix

Creating the tf-idf matrix is as simple as passing the *freq_term_matrix* we defined above into TfidfTransformer's fit method.

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidf = TfidfTransformer(norm=”l2")
tfidf.fit(freq_term_matrix)
```

Once we have the tf-idf transformer fitted, we can take the original document, vectorize it, and transform it into a tf-idf matrix.

```
doc_freq_term = count_vect.transform([doc])
doc_tfidf_matrix = tfidf.transform(doc_freq_term)
```

Note: The output of **doc_tfidf_matrix** will be a matrix with a single row because we have only passed in one document.

it's as simple as looking up the index value for each word in a sentence and finding the tf-idf score in **doc_tfidf_matrix**.

```
tfidf_sent = [[doc_matrix[feature_names.index(w.lower())]
            for w in sent if w.lower() in feature_names]
            for sent in sentences]
```

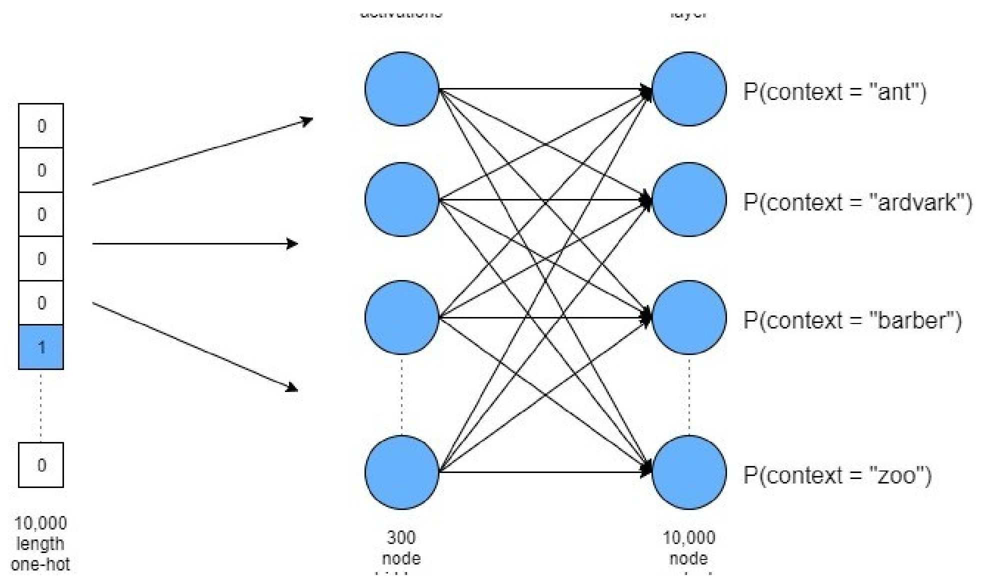
Code to calculate the position weights as below:

```
ranked_sents = [sent*(i+1/len(sent_values))
             for i, sent in enumerate(sent_values)]
```

7. Word2Vec word embedding by using Python and TensorFlow

Word2Vec word embedding using TensorFlow is basically an idea of how to create deep learning models that predict text sequences. The key ideas in NLP is how we can efficiently convert words into numeric vectors which can then be “fed into” various machine learning models to perform predictions.

Consider a word embedding softmax trainer



Need of Word2Vec

If we want to feed words into machine learning models, unless we are using tree based methods, we need to convert the words into some set of numeric vectors. A straight-forward way of doing this would be to use a “one-hot” method of converting the word into a sparse representation with only one element of the vector set to 1, the rest being zero.

Consider the sentence “the cat sat on the mat” we would have the following vector representation as below:

$$\begin{pmatrix} the \\ cat \\ sat \\ on \\ the \\ mat \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

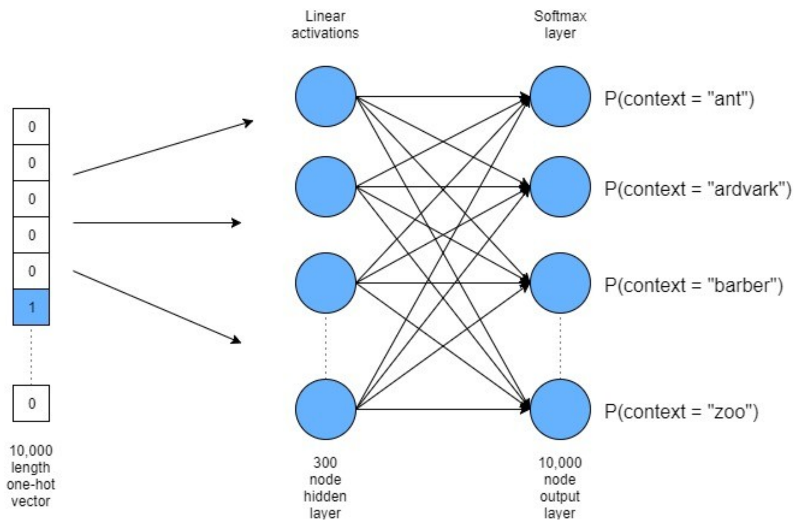
Here we have transformed a six word sentence into a 6×5 matrix, with the 5 being the size of the *vocabulary* (“the” is repeated).

The Word2Vec methodology

There is two components to the Word2Vec methodology. The first is the mapping of a high dimensional one-hot style representation of words to a lower dimensional vector. This might involve transforming a 10,000 columned matrix into a 300 columned matrix, for instance. This process is called word embedding. The second goal is to do this while still maintaining word context and therefore, to some extent, meaning. One approach to achieving these two goals in the Word2Vec methodology is by taking an input word and then attempting to estimate the probability of other words appearing close to that word. This is called the skip-gram approach. The alternative method, called Continuous Bag Of Words (CBOW), does the opposite – it takes some context words as input and tries to find the single word that has the highest probability of fitting that context.

The softmax Word2Vec method in TensorFlow

Consider an example of sentence as ‘The cat sat on the mat’ is part of a much larger text database, with a very large vocabulary – say 10,000 words in length. We can reduce this to a 300 length embedding. Diagram for Word2Vec softmax trainer as below



As with any machine learning problem, there are two components – the first is getting all the data into a usable format, and the next is actually performing the training, validation and testing.

Preparing the text data

```
def maybe_download(filename, url, expected_bytes):
    if not os.path.exists(filename):
        filename, _ = urllib.request.urlretrieve(url + filename, filename)
    statinfo = os.stat(filename)
    if statinfo.st_size == expected_bytes:
        print('Found and verified', filename)
    else:
        print(statinfo.st_size)
        raise Exception(
            'Failed to verify ' + filename + '. Can you get to it with a browser?')
    return filename
```

To call the function with the data-set we are using in this example, we execute the following code:

```
url = 'http://mattmahoney.net/dc/'
filename = maybe_download('text.zip', url, 31344016)
```

The next thing we have to do is take the *filename* object, which points to the downloaded file, and extract the data using the Python *zipfile* module.

```
# Read the data into a list of strings.
def read_data(filename):
    with zipfile.ZipFile(filename) as f:
        data = tf.compat.as_str(f.read(f.namelist()[0])).split()
    return data
```

Finally, we use *split()* function to create a list with all the words in the text file, separated by white-space characters. We can see some of the output here:

```
vocabulary = read_data(filename)
print(vocabulary[:7])
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse']
```

The returned vocabulary data contains a list of plain English words, ordered as they are in the sentences of the original extracted text file.

```
def build_dataset(words, n_words):
    """Process raw inputs into a dataset."""
    count = [['UNK', -1]]
    count.extend(collections.Counter(words).most_common(n_words - 1))
    dictionary = dict()
    for word, _ in count:
        dictionary[word] = len(dictionary)
    data = list()
    unk_count = 0
    for word in words:
        if word in dictionary:
            index = dictionary[word]
        else:
            index = 0  # dictionary['UNK']
            unk_count += 1
        data.append(index)
    count[0][1] = unk_count
    reversed_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return data, count, dictionary, revers
```

The next part of this function creates a dictionary, called *dictionary* which is populated by keys corresponding to each unique word. The value assigned to each unique word key is simply an increasing integer count of the size of the dictionary. So, for instance, the most common word will receive the value 1, the second most common the value 2, the third most common word the value 3, and so on (the integer 0 is assigned to the ‘UNK’ words). This step creates a unique integer value for each word within the vocabulary – accomplishing the second step of the process which was defined above. Now to create a data set comprising of our input words and associated grams, which can be used to train our Word2Vec embedding system. The code to do this is:

```
data_index = 0
# generate batch data
def generate_batch(data, batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size), dtype=np.int32)
    context = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    span = 2 * skip_window + 1  # [ skip_window input_word skip_window ]
    buffer = collections.deque(maxlen=span)
    for _ in range(span):
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    for i in range(batch_size // num_skips):
        target = skip_window  # input word at the center of the buffer
        targets_to_avoid = [skip_window]
        for j in range(num_skips):
            while target in targets_to_avoid:
                target = random.randint(0, span - 1)
            targets_to_avoid.append(target)
            batch[i * num_skips + j] = buffer[skip_window]  # this is the input word
            context[i * num_skips + j, 0] = buffer[target]  # these are the context words
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    # Backtrack a little bit to avoid skipping words in the end of a batch
    data_index = (data_index + len(data) - span) % len(data)
    return batch, context
```

In the above function first the batch and label outputs are defined as variables of size *batch_size*. Then the span size is defined, which is basically the size of the word list that the input word and context samples will be drawn from. In the example sub-sentence above "the cat sat on the", the span is 5 = 2 x skip window + 1. After this a buffer is created:

```
buffer = collections.deque(maxlen=span)
for _ in range(span):
    buffer.append(data[data_index])
    data_index = (data_index + 1) % len(data)
```

The position of the buffer in the input text stream is stored in a global variable *data_index* which is incremented each time a new word is added to the buffer. If it gets to the end of the text stream, the "% len(data)" component of the index update will basically reset the count back to zero.

The code below fills out the batch and context variables:

```
for i in range(batch_size // num_skips):
    target = skip_window  # input word at the center of the buffer
    targets_to_avoid = [skip_window]
    for j in range(num_skips):
        while target in targets_to_avoid:
            target = random.randint(0, span - 1)
        targets_to_avoid.append(target)
        batch[i * num_skips + j] = buffer[skip_window]  # this is the input word
        context[i * num_skips + j, 0] = buffer[target]  # these are the context words
    buffer.append(data[data_index])
    data_index = (data_index + 1) % len(data)
```

Now consider the code below shows how to grab some random validation words from the most common words in our vocabulary:

```
# construction are also the most frequent.
valid_size = 16
valid_window = 100  # Only pick dev samples in the head of the distribution.
valid_examples = np.random.choice(valid_window, valid_size, replace=False)
```

Creating the TensorFlow model

The first thing to do is set-up some variables which can be use later on in the code:

```
batch_size = 128
embedding_size = 128  # Dimension of the embedding vector.
skip_window = 1       # How many words to consider left and right.
num_skips =
```

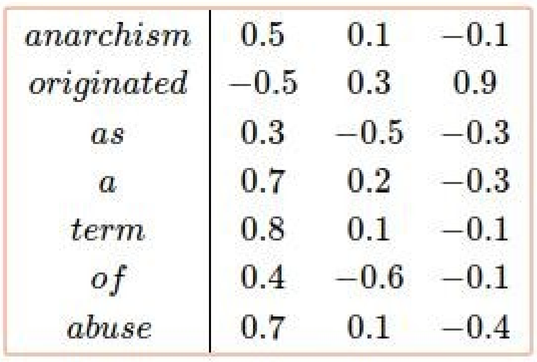
Next we setup some TensorFlow placeholders that will hold our input words and context words which we are trying to predict. We also need to create a constant to hold our validation set indexes in TensorFlow:

```
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
```

Next, we need to setup the embedding matrix variable / tensor – this is straight-forward using the TensorFlow *embedding_lookup()* function

```
# Look up embeddings for inputs.
embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

Here's a simplified example (using dummy values), where *vocabulary_size=7* and *embedding_size=3*:



anarchism	0.5	0.1	−0.1
originated	−0.5	0.3	0.9
as	0.3	−0.5	−0.3
a	0.7	0.2	−0.3
term	0.8	0.1	−0.1
of	0.4	−0.6	−0.1
abuse	0.7	0.1	−0.4

Next we have to create some weights and bias values to connect the output softmax layer, and perform the appropriate multiplication and addition. This looks like:

```
# Construct the variables for the softmax
weights = tf.Variable(tf.truncated_normal([vocabulary_size, embedding_size],
                          stddev=1.0 / math.sqrt(embedding_size)))
biases = tf.Variable(tf.zeros([vocabulary_size]))
hidden_out = tf.matmul(embed, tf.transpose(weights)) + biases
```

Now the code below performs gradient descent optimization operation as

```
# convert train_context to a one-hot format
train_one_hot = tf.one_hot(train_context, vocabulary_size)
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=hidden_out,
    labels=train_one_hot))
# Construct the SGD optimizer using a learning rate of 1.0.
optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(cross_entropy)
```

First, we calculate the L2 norm of each vector using the *tf.square()*, *tf.reduce_sum()* and *tf.sqrt()* functions to calculate the square, summation and square root of the norm, respectively:

```
# Compute the cosine similarity between minibatch examples and all embeddings.
norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims=True))
normalized_embeddings = embeddings / norm
```

Now we can look up our validation words / vectors using the *tf.nn.embedding_lookup()* that we discussed earlier:

```
valid_embeddings = tf.nn.embedding_lookup(
      normalized_embeddings, valid_dataset)
```

Now that we have the normalized validation tensor, *valid_embeddings*, we can multiply this by the full normalized vocabulary (*normalized_embedding*) to finalize our similarity calculation:

```
similarity = tf.matmul(
      valid_embeddings, normalized_embeddings, transpose_b=True)
```

Running the TensorFlow model

```
with tf.Session(graph=graph) as session:
  # We must initialize all variables before we use them.
  init.run()
  print('Initialized')

  average_loss = 0
  for step in range(num_steps):
    batch_inputs, batch_context = generate_batch(data,
        batch_size, num_skips, skip_window)
    feed_dict = {train_inputs: batch_inputs, train_context: batch_context}

    # We perform one update step by evaluating the optimizer op (including it
    # in the list of returned values for session.run()
    _, loss_val = session.run([optimizer, cross_entropy], feed_dict=feed_dict)
    average_loss += loss_val

    if step % 1000 == 0:
      if step > 0:
        average_loss /= 1000
      # The average loss is an estimate of the loss over the last 2000 batches.
      print('Average loss at step ', step, ': ', average_loss)
      average_loss = 0
```

Next, we want to print out the words which are most similar to our validation words – we do this by calling the similarity operation we defined above and sorting the results as below:

```
# Note that this is expensive
if step % 10000 == 0:
    sim = similarity.eval()
    for i in range(valid_size):
        valid_word = reverse_dictionary[valid_examples[i]]
        top_k = 8  # number of nearest neighbors
        nearest = (-sim[i, :]).argsort()[1:top_k + 1]
        log_str = 'Nearest to %s:' % valid_word
        for k in range(top_k):
            close_word = reverse_dictionary[nearest[k]]
            log_str = '%s %s,' % (log_str, close_word)
        print(log_str)
```

This function first evaluates the similarity operation, which returns an array of cosine similarity values for each of the validation words. Then we iterate through each of the validation words, taking the top 8 closest words by using argsort() on the negative of the similarity to arrange the values in descending order. The code then prints out these 8 closest words so we can monitor how the embedding process is performing.

Finally, after all the training iterations are finished, we can assign the final embeddings to a separate tensor for use late.

```
final_embeddings = normalized_embeddings.eval()
```

Speeding things

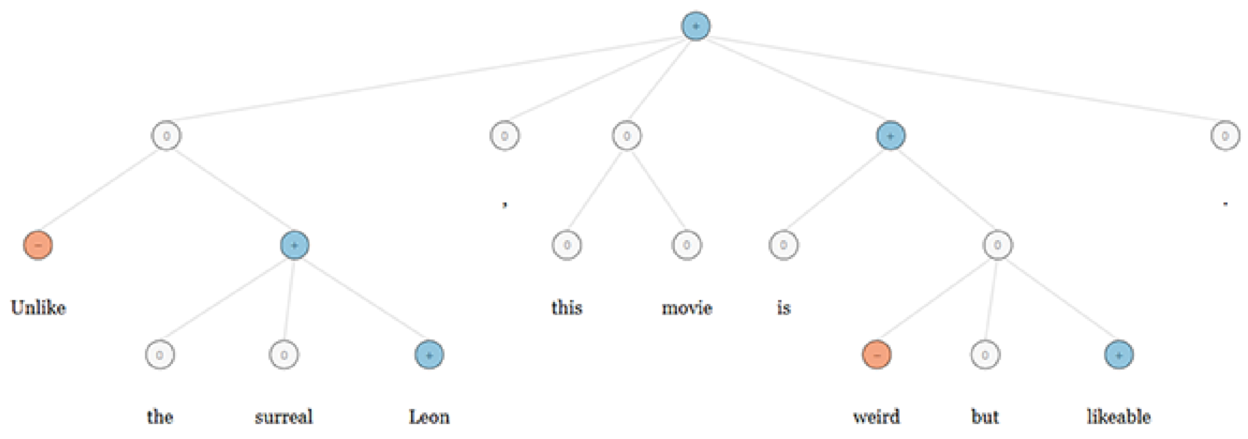
TensorFlow has helped us out here, and has supplied an NCE loss function that we can use called *tf.nn.nce_loss()* which we can supply weight and bias variables to.

```
# Construct the variables for the NCE loss
nce_weights = tf.Variable(
        tf.truncated_normal([vocabulary_size, embedding_size],
                            stddev=1.0 / math.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))

nce_loss = tf.reduce_mean(
        tf.nn.nce_loss(weights=nce_weights,
                       biases=nce_biases,
                       labels=train_context,
                       inputs=embed,
                       num_sampled=num_sampled,
                       num_classes=vocabulary_size))

optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(nce_loss)
```

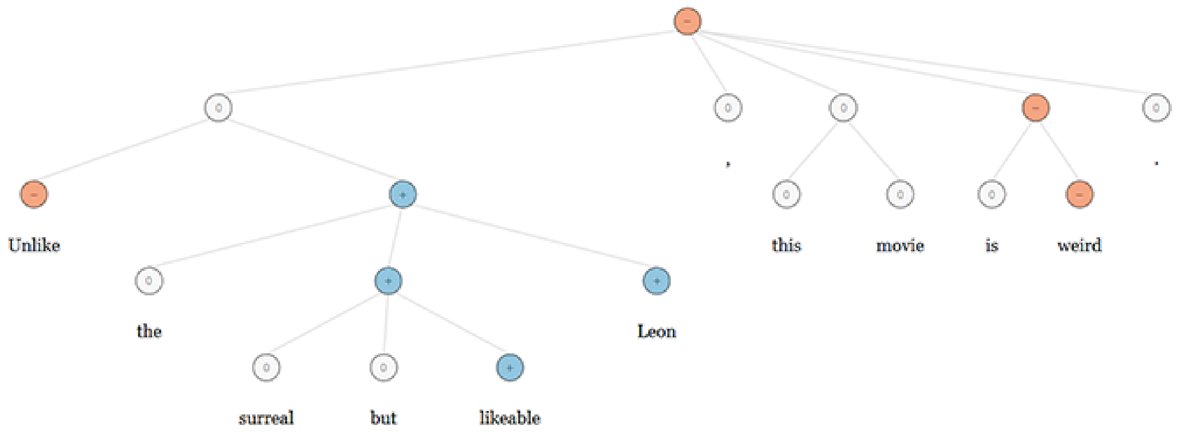
8. Sentiment Analysis Using Python

The Most Widely Used Methods Of Sentiment Analysis Have Been Limited To So-called ‘bag Of Words’ Models, Which Don’t Take Word Order Into Account. They Simply Parse Through A Collection Of Words, Mark Each As Positive Or Negative, And Use That Count To Estimate Whether A Sentence Or Paragraph Has A Positive Or Negative Meaning.

In Evaluating Sample Sentences, NaSent Organizes Words Into What The Scientists Call Grammatical Tree Structures (as Per Image Below) That Put The Words Into Contex



For example, NaSent’s analysis of the sentence: ‘Unlike the surreal Leon, this movie is weird but likeable.’Red nodes indicate that the model assigned a negative sentiment to the underlying word or phrase, such as in the case of ‘weird.’ The phrase ‘is weird but likeable’is correctly assigned a positive sentiment. Similarly, in NaSent’s analysis of ‘Unlike the surreal but likeable Leon, this movie is weird.’Sentence, the model correctly focuses on the overall negative sentiment, even though this sentence uses exactly the same words as the one above. This is represented by the grammatical tree structure as per image below.



```
#import necessary modules
from twython import Twython
import sys
import os
import time
import sqlite3
import re

#set up some of the variables/lists
db_filename = r"c:\Twitter_Test_Data_Folder\temp_db" #database file name
consumer_key = ""  #application's consumer key
consumer_secret = "" #application's consumer secret
access_token = "" #application's access_token
access_token_secret = ""  #application's access_token_secret
harvest_keyword = 'Microsoft' #searches Twitter feeds with this keyword
file_location = r'c:\Twitter_Test_Data_Folder\Twitter_Test_Data.txt'
exec_times = 100  #number of times Twitter feeds will be fetched
pause_for = 15   #sleep for 30 seconds between intervals

#define database table and database dropping query
create_schema = "CREATE TABLE twitter_feeds \
                (id integer primary key autoincrement not null,\
                twitter_feed text)"
drop_schema = "DROP TABLE twitter_feeds"

#create database file and schema using the scripts above
db_is_new = not os.path.exists(db_filename)
with sqlite3.connect(db_filename) as conn:
    if db_is_new:
        print("Creating temp database schema on " + db_filename + " database ...\n")
        conn.executescript(create_schema)
    else:
        print("Database schema may already exist. Dropping database schema on " + db_filename +
"...")
        #os.remove(db_filename)
        conn.executescript(drop_schema)
        print("Creating temporary database schema...\n")
        conn.executescript(create_schema)
```

Fetch Twitter feeds and format them, removing unwanted characters/strings

```
def fetch_twitter_feeds():
    twitter = Twython(consumer_key, consumer_secret, access_token, access_token_secret)
    search_results = twitter.search(q=harvest_keyword, rpp="100")
    for tweet in search_results['statuses']:
        try:
            #the following can be enabled to see Twitter feeds content being fetched
            #print("-" * 250)
            #print("Tweet from @%s -->" % (tweet['user']['screen_name']))
            #print("    ", tweet['text'], "\n")
            #print("-" * 250)
            feed = tweet['text']
            feed = str(feed.replace("\n", ""))                    #concatnate if tweet is on multiple
lines
            feed = re.sub(r'http://[\w.]+/+[\w.]+', "", feed, re.IGNORECASE)    #remove http:// URL
shortening links
            feed = re.sub(r'https://[\w.]+/+[\w.]+',"", feed, re.IGNORECASE)    #remove https:// URL
shortening links
            feed = re.sub('[@#$<>:%&]', '', feed)                 #remove certain characters
            cursor = conn.cursor()
            cursor.execute("INSERT INTO twitter_feeds (twitter_feed) SELECT (?)", [feed])
#populate database table with the feeds collected
        except:
            print("Unexpected error:", sys.exc_info()[0])
            conn.rollback()
        finally:
            conn.commit()
```

Now delete duplicated, too short or empty records from 'twitter_feeds' table as

```
def drop_dups_and_short_strs():
    try:
        cursor = conn.cursor()
        cursor.execute("DELETE  FROM twitter_feeds WHERE id NOT IN(\
                        SELECT  MIN(id) FROM  twitter_feeds GROUP BY twitter_feed)")
        cursor.execute("DELETE FROM twitter_feeds WHERE LENGTH(twitter_feed) < 10")
        cursor.execute("DELETE FROM twitter_feeds WHERE twitter_feed IS NULL OR
twitter_feed = "")
    except:
        print("Unexpected error:", sys.exc_info()[0])
        conn.rollback()
    finally:
        conn.commit()
```

display progress bar in a console

```
def progressbar(it, prefix = "", size = 60):
    count = len(it)
    def _show(_i):
        x = int(size*_i/count)
        print("%s[%s%s] %i/%i\r"  % (prefix, "#"*x, "."*(size-x), _i, count), end='')
        sys.stdout.flush()
    _show(0)
    for i, item in enumerate(it):
        yield item
        _show(i+1)
    print()
```

Finally main() method will execute

```
if __name__ == '__main__':
    try:
        for i, z in zip(range(exec_times+1), progressbar(range(exec_times), "Fetching Twitter Feeds:
", 100)):
            fetch_twitter_feeds()
            time.sleep(pause_for)
        drop_dups_and_short_strs()
        cursor = conn.cursor()
        tweets = cursor.execute("SELECT twitter_feed tw FROM twitter_feeds")
        f = open(file_location, 'w', encoding='utf-8')
        for row in tweets:
            row = ''.join(row)
            f.write(row)
            f.write("\n")
    except:
        print("Unexpected error:", sys.exc_info()[0])
    finally:
        conn.close()
        f.close()
        os.remove(db_filename)
```

CONCLUSION

Hence Sentiment analysis is becoming a popular area of research and social media analysis, especially around user reviews and tweets. It is a special case of text mining generally focused on identifying opinion polarity, and while it’s often not very accurate, it can still be useful.

9. GloVe: Global Vectors for Word Representation using python

Semantic vector space models of language represent each word with a real-valued vector. These vectors can be used as features in a variety of applications, such as information retrieval, document classification, question answering, named entity recognition, and parsing. Most word vector methods rely on the distance or angle between pairs of word vectors as the primary method for evaluating the intrinsic quality of such a set of word representations.

The word vector space by examining not the scalar distance between word vectors, but rather their various dimensions of difference. For example, the analogy “king is to queen as man is to woman” should be encoded in the vector space by the vector equation king − queen = man − woman. This evaluation scheme favors models that produce dimensions of meaning, thereby capturing the multi-clustering idea of distributed representations.

Glove Model

The statistics of word occurrences in a corpus is the primary source of information available to all unsupervised methods for learning word representations, and although many such methods now exist, the question still remains as to how meaning is generated from these statistics, and how the resulting word vectors might represent that meaning. In this section, we shed some light on this question. We use our insights to construct a new model for word representation which we call GloVe, for Global Vectors, because the global corpus statistics are captured directly by the model.

Glove Model Implementation using python

Python code:

```
from gensim import utils, corpora, matutils, models
import glove

# Restrict dictionary to the 30k most common words.
wiki = models.word2vec.LineSentence('/data/shootout/title_tokens.txt.gz')
id2word = corpora.Dictionary(wiki)
id2word.filter_extremes(keep_n=30000)
word2id = dict((word, id) for id, word in id2word.iteritems())

# Filter all wiki documents to contain only those 30k words.
filter_text = lambda text: [word for word in text if word in word2id]
filtered_wiki = lambda: (filter_text(text) for text in wiki)  # generator

# Get the word co-occurrence matrix -- needs lots of RAM!!
cooccur = glove.Corpus()
cooccur.fit(filtered_wiki(), window=10)

# and train GloVe model itself, using 10 epochs
model_glove = glove.Glove(no_components=600, learning_rate=0.05)
model_glove.fit(cooccur.matrix, epochs=10)
```

10. Language Identification from Texts using Bi-gram model using Python/NLTK

There are a lot of social websites (Facebook, Twitter, etc), messengers and applications that support various languages in which we can write. In order to process these texts, one must identify the language of the text before further processing or performing machine translation (translating text from one language to other). Language identification is formulated as a supervised machine learning task of mapping a text to a unique language from a set of trained languages. It involves building language models from a lot of text data of respective languages and then identifying the test data (text) among the trained language models. So, lets get started with building the language models.

There are following steps are involved as follows

Building Bi-gram Language model

Bi-gram language model is created for each of the six languages. Finding bi-grams and their frequencies will be achieved through NLTK (Natural language toolkit) in Python. NLTK is a popular open source toolkit, developed in Python for performing various tasks in text processing.

Pre-processing the text corpus

We have considered the following pre-processing steps before creating bi-gram language model.

1. All the texts were converted to lower case.
2. All the digits were removed from the text sentences.
3. Punctuation marks and special characters were removed.
4. All the sentences were concatenated with space in between.
5. Series of contiguous white spaces were replaced by single space.

It is important to note that the text file must be read in Unicode format (UTF-8 encoding) which encompasses the character set including all the languages.

Bi-gram extraction and Language model

Now, we will use library functions of NLTK to find out list of bi-grams sorted with number of occurrences for each language.

```
finder = BigramCollocationFinder.from_words(seq_all)
finder.apply_freq_filter(5)
bigram_model = finder.ngram_fd.viewitems()
bigram_model = sorted(finder.ngram_fd.viewitems(), key = lambda item : item[1], reverse = True)
```

The full Python implementation of building a language model from text corpus using NLTK is as below.

```
from nltk.collocations import BigramCollocationFinder
import re
import codecs
import numpy as np
import string
```

```
def train_language(path,lang_name):
   words_all = []
   translate_table = dict((ord(char), None) for char in string.punctuation)
   # reading the file in unicode format using codecs library
   with codecs.open(path,"r","utf-8
      for i,line in enumerate(filep):
         # extracting the text sentence from each line
         line = " ".join(line.split()[1:])
         line = line.lower()   # to lower case
         line = re.sub(r"\d", line) # remove digits

         if len(line) != 0:
            line = line.translate(translate_table) # remove punctuations
            words_all += line
            words_all.append(" ") # append sentences with space

   all_str = ''.join(words_all)
   all_str = re.sub(' +',' ',all_str) # replace series of spaces with single space
   seq_all = [i for i in all_str]

   # extracting the bi-grams and sorting them according to their frequencies
   finder = BigramCollocationFinder.from_words(seq_all)
   finder.apply_freq_filter(5)
   bigram_model = finder.ngram_fd.viewitems()
   bigram_model = sorted(finder.ngram_fd.viewitems(), key=lambda item: item[1],reverse=True)

   print bigram_model
   np.save(lang_name+".npy",bigram_modelanguage model
```

```
if __name__ == "__main__":
   root = "train\\"
   lang_name = ["french","english","german","italian","dutch","spanish"]
   train_lang_path = ["fra_news_2010_30K-text\\fra_news_2010_30K-
sentences.txt","eng_news_2015_30K\\eng_news_2015_30K-
sentences.txt","deu_news_2015_30K\\deu_news_2015_30K-sentences.txt","ita_news_2010_30K-
text\\ita_news_2010_30K-sentences.txt","nld_wikipedia_2016_30K\\nld_wikipedia_2016_30K-
sentences.txt","spa_news_2011_30K\\spa_news_2011_30K-sentences.txt
train_language(root+p,lang_name[i])
```

The above code includes all the steps of pre-processing and creating the bi-gram language model of each language in train directory.

Evaluation on Test Corpus

To classify a text sentence among the language models, the distance of the input sentence is calculated with the bi-gram language model. The language with the minimal distance is chosen as the language of the input sentence. Once the pre-processing of the input sentence is done, the bi-grams are extracted from the input sentence.

The full implementation of closed set evaluation of language identification task on wortschatz test corpus is as below.

```
if __name__ == "__main__":
    root = "test\\"
    lang_name = ["english","german","french","italian","dutch","spanish"]

    no_of_bigms = []
    for i,lang in enumerate(lang_name):
        model = np.load(lang+".npy      total = 0
        for key,v in model:
            total = total + v
        no_of_bigms.append(total)
        print total

    train_lang_path = ["eng_news_2015_10K\\eng_news_2015_10K-
sentences.txt","deu_news_2015_10K\\deu_news_2015_10K-sentences.txt","fra_news_2010_10K-
text\\fra_news_2010_10K-sentences.txt","ita_news_2010_10K-text\\ita_news_2010_10K-
sentences.txt","nld_wikipedia_2016_10K\\nld_wikipedia_2016_10K-
sentences.txt","spa_news_2011_10K\\spa_news_2011_10K-sentences.txt    print "Testing of
",lang_name[i]
        test_language(root+p,lang_name[i],no_of_bigms)
```

Evaluation on LIGA Twitter dataset

LIGA dataset contains 9066 pre-processed and cleaned tweets. I have tested these tweets against the language models created earlier. One does not need to do any pre-processing as these tweets are already clean. The evaluation code for Twitter dataset has been left for readers to write as it will be similar to the evaluation code shown earlier for test corpus. Similar to confusion matrix of test corpus, confusion matrix for LIGA dataset has been shown below.

```
LanguagesEnglishGermanFrenchItalianDutchSpanish
English11772796379474
German18124531171599
French7151421421947
Italian5371484436
Dutch112761013715
Spanish282057161459
```