Detecting Semantic Change

Alexander Escorcio
Supervisor: Dr. Martin Emms
Final Year Project

School of Computer Science and Statistics
Trinity College Dublin

April 7, 2015
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

__________________________________________ April 7, 2015

Alexander Escorcio
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_________________________________________ April 7, 2015

Alexander Escorcio
Acknowledgements

I would like to thank Dr. Martin Emms for his guidance and time during the completion of this project, and I would like to extend this word to Dr. Carl Vogel and the other inspired professors of these disciplines. Many kind words have been written about many fine teachers, and all of them are true.

Reckoning with change can push thoughts in the direction of things that never do. This work is due to my parents, my brothers, my family, and my friends.
Abstract

Words can change semantically, gaining or losing meanings and warping in sense over time. The word *wood* currently refers to a certain material or to a gathering of trees, though this might seem quaint to future readers. Word Sense Disambiguation and Word Sense Induction are fields concerned with distinguishing which sense of a polysemous word, like *wood*, is meant in discourse. This project explores means of performing such disambiguation using the Expectation Maximisation and K-Means algorithms on untagged corpora in conjunction with various established and one proposed dimensionality reduction technique. Subsequently, it is seen that these methods can be used to detect semantic change in language, and this property is also demonstrated.

Theory

1 Word Sense Disambiguation and Induction

1.1 Introduction

In language, it is not uncommon for the production of a word in isolation to have more than one potential interpretation; in both written and spoken English, the word *light* can have any one of a number of meanings. For example, it can be used as a noun to refer to a source of illumination, as an adjective to describe something as being the opposite of heavy, as a verb meaning to ignite, and so on.

This ambiguity is by no means restricted to written and spoken language; its presence elsewhere may be attested to through its existence in sign language, for example. In American Sign Language (ASL), the sign for *wagging tail*, is precisely the same as the sign for *where* (Sutton-Spence, 2005, pp.88-89).

As such, the English word *light* and the ASL sign mentioned above may both be correctly termed as polysemous, having more than one meaning, and are inherently ambiguous in sense.

Word Sense Disambiguation (WSD) is an area of natural language processing that is concerned with being able to evaluate which of a word’s meanings is intended in a particular utterance. Consider these two extracts from Vladimir Nabokov’s *Lolita*:

(1) I have turned on the light to take down a dream.

(2) Haze suddenly spoiled everything by turning to me and asking me for a light.

In sentence (1), a successful WSD of the polysemous *light* would recognize that it was intended to denote the nominal form of the word relating to a light source. In sentence (2), a successful WSD would categorise *light* as being a noun referring to a lighter.

Word Sense Induction (WSI) is the process of grouping similar meanings, without being concerned about the semantic properties of the groups. If WSI
were carried out on (1) and (2) with respect to the word light, a successful result would state that there are two senses of light present in the data, one in (1) and another in (2). WSI would not offer any further estimations.

1.2 Beginnings

Incarnations of WSD and WSI have been topics of interest in the field of linguistics since at least 1949, as evidenced by Warren Weaver’s (1949) writing in his memorandum on machine translation:

“If one examines the words in a book, one at a time as through an opaque mask with a hole in it one word wide, then it is obviously impossible to determine, one at a time, the meaning of the words.”

Likewise, suggested methods for mechanically disambiguating a word’s sense have been present at least since that same publication, with Warren suggesting that “if one lengthens the slit in the opaque mask, until one can see not only the central word in question, but also say N words on either side, then if N is large enough one can unambiguously decide the meaning of the central word.” Warren’s early slitted mask allusion remains pertinent today, with the majority of WSD and WSI deriving information from a polysemous word’s context (Mitkov, 2003).

1.3 Contexts

In this document, the term context is used to refer to the range R of words that appear on either side of (i.e., co-occur with, are in the window of) a particular word.

For example, considering a context of size R=2 for the word light in (1), the co-occurring words are ‘on’, ‘the’, ‘to’ and ‘take’; two words to the left and two words to the right of ‘light’.

(3) ... on the light to take ...
The context where R=2 for light in example (1).

With the small context given in (3), a human challenged to identify the sense of the word light might hesitate to place lofty bets on any one particular meaning. However, extending the range of the context to R=3 renders:

(4) ... turned on the light to take down ...
The context where R=3 for light in example (1).

Here, due to the inclusion of the telling phrasal verb construction, the sense of the word light as an illuminating device becomes much more likely.

1.4 Strategies

In the past, approaches to WSD and WSI have hinged on a variety of context manipulations. Mitkov (2003) notes that “Wilks (1972) made one of the earliest attempts to solve the WSD problem”, doing so by attributing certain formulae to items in the context of an ambiguous word, and choosing the highest scoring formula for the entire string with respect to various semantic relationships between words.
Other such approaches, similarly reliant on having a pre-established reserve of linguistic information in a lexicon, continued into the 1980s. In 1986, Lesk devised a method of word-sense disambiguation using “machine readable dictionaries, and looking for words in the sense definitions that overlap words in the definition of nearby words.” (Lesk, 1986). The refinement of this technique has since been the subject of a number of works, predominantly in the 1990s (Mitkov, 2003).

In 1991, Brown et al. employed statistical and machine learning techniques to align sentences in parallel corpora, achieving a high success rate without being challenged by the constraints imposed by requiring lexical information on words being processed.

Drawing from analogous experiments, which considered the probabilities of context words given a certain sense of the polysemous word, Yarowsky et al. (1992) observed that “if a polysemous word such as sentence appears two or more times in a well-written discourse, it is extremely likely that they will all share the same sense.” This notion was furthered a year later to put forward that any polysemous word occurring more than once in the same collocation is highly likely to have the same sense in each occurrence within the collocation (Yarowsky, 1993). Yarowsky made use of his maxims in a 1995 work that combined an unsupervised learning algorithm for WSD with a sense classifier that factored in the one sense per discourse and one sense per category accessions. The work is entitled Unsupervised word sense disambiguation rivaling supervised methods.

1.5 Supervised and Unsupervised Machine Learning

The field of Machine Learning is concerned with creating algorithms that facilitate an agent’s (a computer’s, for instance) improving performance of a task.

Mitchell (1997) defines learning as follows: “[An agent] is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with E.”

For the most part, learning algorithms part into two flavours: supervised and unsupervised. Simply put, a machine learning algorithm that uses annotated data is supervised, and a machine learning algorithm that does not used annotated data is unsupervised.

As described above, WSD and WSI used mainly supervised methods in the fields’ early years. However, annotated corpora are not ample, and as such, unsupervised methods have received more attention recently. When it comes to detecting semantic change, unsupervised methods of WSD have the keen advantage of being able to work on data as soon as it is available. To this end, unsupervised learning algorithms were chosen as the means of clustering word senses in this project.
2 Semantic Change

2.1 Introduction

When a word acquires a new meaning or loses an old one, it can be said to have undergone a semantic change. As words are not restricted to one acquisition or loss at a time, ambiguity in what they denote can arise.

With accreditation to a number of semanticists, Narrog (2012, pp.61) puts it that “Linguistic forms [...] normally don’t start out being polysemous. Rather, in the course of their historical development, new meanings develop and branch off from the original meaning. Polysemy has therefore long been thought of as a product of the process of meaning change and meaning extension.”

Mechanisms that are proposed to account for semantic change, while varying in terminology amongst voices in the field, are said by Narrog (2012, pp.62) to generally include metaphorization, metonymization, broadening, narrowing, amelioration, and pejoration.

Metaphor describes the attainment of a sense by allusion. For example, the word *mouse*, just prior to the existence of the mechanical device, was used mainly to denote a type of animal. By the analogy to the small and often grey creature, the small and often grey device also became termed *mouse*. (Harper, 2015).

Metonymization in semantic change is when a word evolves to refer to something it is related to. For example, the use of the word *screen* to mean *monitor*.

Amelioration is a word’s adoption of a more positive sense, and pejoration is a word’s adoption of a more negative sense. Consider the usages, contrasting in positivity, of the word *crazy* in the following excerpts from Kerouac’s *On The Road*:

(5) If you keep this up you’ll both go crazy, but let me know what happens as you go along.

(6) I felt like a million dollars; I was adventuring in the crazy American night. Example (6) demonstrates the relatively recent amelioration of the word.

Under the entry for *meat* in *The Concise Oxford dictionary of English etymology*, it is noted that, in Old English, *meat* referred not only to the flesh of an animal, but to food in general. (Hoad, 1993). Thus, a semantic narrowing of the word *meat* has since occurred. A semantic broadening is the opposite of this. The word *holiday* originally evoked the sense of a “religious festival”, a holy day. (Hoad, 1993). Today, its sense is broadened to mean any period of relaxation.

2.2 Detecting Semantic Change

With any genre of semantic change, a shift in the words that commonly occur alongside the object of the change is affected.

Considering the word *mouse* as a referent to the rodent, one might expect to find co-occurring words like *field, cheese, fur,* etc. However, where *mouse* is used to denote the mechanical device that often accompanies a personal
computer, field, cheese and fur would be less likely to occur within its context. Rather, words like pointer, keyboard and computer would be more common.

Let us imagine that, for every year between 1920 and 1990, we have a corpus of 1000 randomly sampled usages of the word mouse, along with a context window of size $R=10$ co-occurrences around each instance of mouse. Now, let us picture a graph whose $x$-axis represents the years from 1920 to 1990, and whose $y$-axis records frequency. For each year, we could plot one point for the number of co-occurrences of mouse with field, cheese or fur, and we could plot another point for the number of co-occurrences of mouse with pointer, keyboard or computer.

Given that mouse was first used in the mechanical sense in the 1960s (Harper, 2015), joining each set of plots, one might expect to obtain something like the following chart:

![Graph](image)

In this synthetic case, co-occurrences of field, cheese or fur remain quite constant throughout. Co-occurrences of pointer, keyboard or computer with
mouse are non-existent until the invention of the pointing device, and then increase steadily as the information age weighs in. The detection of semantic change can be reached through a mechanical recognition of this phenomenon. Precisely stated, a significant change in the context of a particular word implies a change in that word’s meaning. Were a computer program able to record the likelihood of each colocate in a word’s context over a range of years, WSI algorithms could then be employed to determine how many senses of the word were present, in what quantities, and in which contexts they occurred. Subsequently, a variance in the WSI results over time could be used to signal a semantic change in the word. In 2013, an example of such a system was implemented. It took advantage of Google’s time-specific search facility to use varying colocate probabilities in detecting semantic change. (Emms, 2013).
Methods

3 The System

3.1 Gathering Data

The aim of this project was to develop a WSI algorithm and apply it to timestamped corpora of contexts for semantically changing and polysemous words. As such, the variance of the WSI results could be examined to reveal trends in the likelihood of one meaning over another, or used to identify the loss or attainment of senses in a word.

Before any processing of data can be done, it is requisite to obtain some data. For the purposes of developing the program, two contrasting texts were processed. With the intuition that two senses of the word rock, one referring to a genre of music and the other referring to a type of mineral, would serve well as the object keywords for word sense induction, two online resources were obtained. The first, (Norton, 2012), is an ebook on the topic of geology. The second, (Stephens, Lamar, 2005), is a dissertation concerning rock music. Some pre-processing of these texts was carried out. Punctuation, with the exception of apostrophes and spaces, was removed. The question of whether to replace capitalized letters with minuscules arose, contention provided by circumstances like White House vs white house. Despite the fact that such majuscules can imply a particular meaning, text was eventually converted to lowercase throughout, avoiding the need for a separate method to tag proper nouns, etc.

Next, a program parsed the texts separately, generating two corpora. The first corpus consisted of contexts co-occurring with rock from the first text, and the second consisted of contexts co-occurring with rock from the second text.

<table>
<thead>
<tr>
<th>Context</th>
<th>Corpus One</th>
<th>Corpus Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>historical views of rock music as socially ledges. Here the rocks lie in horizontal layers</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>early African American rock performers and the foundation of solid rock which everywhere is closely at the rock we see that</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>executive levels of rock production and distribution</td>
<td></td>
</tr>
</tbody>
</table>

3.2 The Expectation Maximisation Algorithm

With context data for two meanings of a particular polysemous word, it was decided that an unsupervised learning algorithm would be employed in WSI. As explored in Part 1, an unsupervised algorithm permits working with what
is essentially raw data, and can be equally as effective as supervised means (Yarowsky, 1995). 

The expectation maximisation, or EM, algorithm “enables parameter estimation in probabilistic models with incomplete data.” (Do, Batzoglou, 2008). In our case, the incomplete, or hidden, data is the meaning of the polysemous word in each context in the corpora.

The EM algorithm consists of two steps: the expectation step and the maximization step. These can be explained in a concise manner by envisioning the application of EM in a coin-flipping scenario.

3.3 Understanding An EM Coin Flip

Imagine there are two coins, coin A and coin B, such that A has a particular probability of landing heads-side-up when flipped, and B has a particular probability of landing heads-side-up when flipped. Let us call these probabilities $\theta_A$ and $\theta_B$, respectively. We know neither the value of $\theta_A$, nor the value of $\theta_B$. We are hoping that EM will eventually estimate the values of $\theta_A$ and $\theta_B$. We have data of coin-flip results; ten sets of ten flips where either coin A or coin B is used for the entirety of each set.

<table>
<thead>
<tr>
<th>Set</th>
<th>Coin</th>
<th>Flip 1</th>
<th>Flip 2</th>
<th>...</th>
<th>Flip 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>H</td>
<td>T</td>
<td>...</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>H</td>
<td>H</td>
<td>...</td>
<td>H</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>T</td>
<td>T</td>
<td>...</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>?</td>
<td>H</td>
<td>T</td>
<td>...</td>
<td>H</td>
</tr>
</tbody>
</table>

If it were the case that we knew which coin were used in each set, we could easily work out the maximum likelihood estimate of $\theta_A$ and $\theta_B$. That is, if we knew which sets of flips belonged to coin A, we could easily work out that the apparent probability of heads landing face up after having flipped coin A is equal to the number of heads in flips of coin A divided by the total number of flips of coin A. The word apparent is used here as we can only judge from the data we have; more data with the same coins could show different trends.

$$maximum\ likelihood\ estimate\ of\ \theta_A = \frac{number\ heads\ using\ coin\ A}{total\ number\ of\ flips\ using\ coin\ A}$$

The same is true for coin B.

$$maximum\ likelihood\ estimate\ of\ \theta_B = \frac{number\ heads\ using\ coin\ B}{total\ number\ of\ flips\ using\ coin\ B}$$

Seeing as we don’t know which coin was used in each set, EM’s approach is to use arbitrary starting parameters for $\theta_A$ and $\theta_B$. For example, we could start by assuming that the coins are unweighted, letting $\theta_A$ equal to 0.5 and $\theta_B$ equal to 0.5.

“These probabilities are used to create a weighted training set consisting of all possible completions of the data.” (Do, Batzoglou, 2008). In other words, for each row of data, using the $\theta_A$ and $\theta_B$ values we guessed, we compute the probability that coin A was used and the probability the coin B was used by considering the data results.
Values for the probability of heads given coin A, heads given coin B, tails
given coin A, and tails given coin B are also calculated. These values are then
employed to generate virtual, or expected, counts of heads and tails under coin
A and coin B for each row.
From these virtual corpora of coin tosses, new (and slightly more accurate)
maximum likelihood estimates are taken for $\theta_A$ and $\theta_B$.
The process is repeated, new $\theta_A$ and $\theta_B$ values filling in for the arbitrary
starting parameters in each iteration, until $\theta_A$ and $\theta_B$ converge upon the
settings that make the data most likely.

3.4 Translating a Coin Flip to WSI
In the field of Natural Language Processing, the word *type* can be used to refer
to a particular item, and the word *token* can be used to refer to instances of a
particular item.
(7) And so faintly you came tapping, tapping at my chamber door
In the above excerpt from the 1845 poem, *The Raven* (Poe, 1966), it can be
said that there are 10 types and 11 tokens. Every type except *tapping* occurs
only once, i.e., generates only one token. *Tapping*, which occurs twice,
generates two tokens.
Types in (7): {and, so, faintly, you, came, tapping, at, my, chamber, door}
Tokens in (7): {and, so, faintly, you, came, tapping, tapping, at, my, chamber,
door}
The above coin-toss EM example translates eloquently to WSI. Two different
senses of a polysemous word, meaning A and meaning B, can be made
analogous to coin A and coin B. The *heads or tails* results in any coin toss
 correspond to the tokens that co-occur in the window of the polysemous word.
Let us consider the following context for R=3 in the dataset described in
section 3.1 of *rock* in relation to music.

(8) $w_1$ $w_2$ $w_3$ $w_4$ $w_5$ $w_6$
     historical views of rock music a socially
As EM calculates values for the probability of meaning A, ($M_A$), and the
probability of meaning B, ($M_B$), in all the data, probabilities for each type
given each meaning, $P(w_n|M_x)$, are also calculated.
Specifically, just as EM calculates $P(\text{heads}—\theta_A)$, $P(\text{tails} |\theta_A)$, $P(\text{heads} |\theta_B)$,
and $P(\text{tails} |\theta_B)$ in its expectation step for the coin example, so would EM
calculate $P(w_1|M_A)$, $P(w_2|M_A)$, ..., $P(w_6|M_A)$, $P(w_1|M_B)$, $P(w_2|M_B)$, ..., $P(w_6|M_B)$ in its expectation step in WSI.
Like with the coin example, after a number of iterations of the expectation
and maximisation steps, $M_A$ and $M_B$ will settle upon particular values. As
will the probability of each type given each meaning. These results may then
be used as a means of classifying each context in the corpora as either meaning
A or meaning B of the polysemous word.
Classification is done by finding which meaning out of the possible set of
meanings maximizes the expression:
θ_M \prod_{k=1}^{n} P(w_k|θ_M), where n is the number of tokens in a context string

For example, consider the table below, which displays synthetic EM results for the context of the word *danced* in the collocation: *the amber sun danced in the sky.*

<table>
<thead>
<tr>
<th>the</th>
<th>amber</th>
<th>sun</th>
<th>in</th>
<th>the</th>
<th>sky</th>
</tr>
</thead>
<tbody>
<tr>
<td>θ_A</td>
<td>P(w_1</td>
<td>θ_A)</td>
<td>P(w_2</td>
<td>θ_A)</td>
<td>P(w_3</td>
</tr>
<tr>
<td>0.42</td>
<td>0.121</td>
<td>0.003</td>
<td>0.00234</td>
<td>0.0231</td>
<td>0.121</td>
</tr>
</tbody>
</table>

| θ_B | P(w_1|θ_B) | P(w_2|θ_B) | P(w_3|θ_B) | P(w_4|θ_B) | P(w_5|θ_B) | P(w_6|θ_B) |
|-----|-------|-----|----|-----|-----|-----|
| 0.68 | 0.111 | 0.033 | 0.4323 | 0.0432 | 0.111 | 0.043 |

Here, θ_M \prod_{k=1}^{n} P(w_k|θ_M), where M = A is computed as follows:

θ_A \prod_{k=1}^{6} P(w_k|θ_A) = 0.42 \cdot 0.121 \cdot 0.003 \cdot 0.00234 \cdot 0.0231 \cdot 0.121 \cdot 0.044 \\
= 4.3875472e-11

And for the other possible meaning in θ_M \prod_{k=1}^{n} P(w_k|θ_M), where M = B:

θ_B \prod_{k=1}^{6} P(w_k|θ_B) = 0.68 \cdot 0.111 \cdot 0.033 \cdot 0.4323 \cdot 0.0432 \cdot 0.111 \cdot 0.043 \\
= 2.22027234e-7

As such, the classifier would find that meaning B maximizes the expression, as 2.22027234e-7 is greater than 4.3875472e-11. That is to say: the words in the context of *danced in the amber sun danced in the sky* were, according to EM, more probable of the hidden variable M_B.

When running the EM algorithm, it is necessary to distinguish how many hidden variables are to be evaluated. For word sense induction, if the number of word senses are known, this parameter can be specified by a user. Otherwise, multiple runs of EM can be carried out for different numbers of hidden variables, and the best fitting results can be retained. A full implementation of the EM algorithm in the Python language may be found in the appendices (Appendix B).

### 3.5 The K-Means Algorithm

“Clustering involves dividing a set of data points into non-overlapping groups, or clusters, of points, where points in a cluster are ‘more similar’ to one another than to points in other clusters.” (Faber, 1994).

The K-Means algorithm is a straight-forward method of clustering items in a dataset. Its working is perhaps most easily understood via example.
Let us imagine there is a word for people whose favourite number is between 10 and 20 and whose least favourite number is between 0 and 10. They are called 'foos'. Let us also imagine that people whose preferences are just the opposite, with their favourite number between 0 and 10 and least favourite number between 10 and 20, are called 'bars'. If we surveyed four foos and four bars, plotting each person’s favourite number against their least favourite number, we might obtain the following data and the following plot. Notice that there is no explicit information as to whether a person is a foo or a bar:

<table>
<thead>
<tr>
<th>Person 1</th>
<th>Person 2</th>
<th>Person 3</th>
<th>Person 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Favourite Number</td>
<td>15</td>
<td>19</td>
<td>17</td>
</tr>
<tr>
<td>Least Favourite Number</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Person 5</th>
<th>Person 6</th>
<th>Person 7</th>
<th>Person 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Favourite Number</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Least Favourite Number</td>
<td>15</td>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>

As with the hidden variables in EM, K-Means must be instructed to search for a certain number, k, of clusters.
Were K-Means able to cluster the data into two groups, automatically identifying which points belong to which cluster, the results could be used to classify person 1 through 8 as either a foo or a bar. The first step in the K-Means algorithm is to create some arbitrary 'centroids'. A centroid is an imaginary point in the data to which other points may be compared. The number of centroids corresponds to the k number of clusters. Let us imagine that our algorithm creates centroids at (10,11) and (10,10).

The Euclidean distance between each point in the data and each centroid is then calculated. Points that are closer to centroid 1 are categorized as being a member of group 1. Points that are closer to centroid 2 are categorized as being a member of group 2.
The average (or mean) point between the items in each cluster is then computed. Centroid 1 is moved to the mean of the points in group 1 and centroid 2 is moved to the mean of the points in group 2.

The process is then repeated, the new centroids each time taking the place of the arbitrarily guessed ones, until they stop changing location and the final cluster estimations are made.

<table>
<thead>
<tr>
<th>Favourite Numbers vs Least Favourite Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

In the above table, the data is clustered into two groups, group 1 or group 2. It is noteworthy that whether group 1 corresponds to foos or bars is not expressed by the algorithm.
3.6 K-Means Applied to Word Sense Induction

“Texts cannot be directly interpreted by a classifier or by a classifier-building algorithm. Because of this, an indexing procedure that maps a text [...] into a compact representation of its content needs to be uniformly applied to training, validation, and test documents.” (Sebastiani, 2002).

In order to apply the K-Means algorithm to some data, the data must be represented in such a way that the Euclidean distance (or another distance comparison; cosine similarity, Hamming Distance, etc) between entries may be found.

For this project, context strings were represented as vectors according to the bag of words model. The dimensionality of each vector was equal to the size of the lexicon of types in the corpora.

(10)

<table>
<thead>
<tr>
<th>Corpus 1</th>
<th>Corpus 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sun in the blue sky</td>
<td>The sun in the amber sky</td>
</tr>
</tbody>
</table>

In (10), our two corpora consist of one context entry each. As such, each vector would have dimensions \{the, sun, in, blue, amber, sky\}.

The magnitude of these vector dimensions could be set either to boolean values, \{0,1\}, or frequency values, \{0, 1, 2, ..., n\}, where the frequency would be the number of tokens of that dimension’s type occurring in the string.

To illustrate this, below are the strings from (10) represented as vectors according to boolean and then frequency dimensional magnitudes.

**Boolean magnitudes representations of strings in (10):**

<table>
<thead>
<tr>
<th>Vector for “The sun in the blue sky”</th>
<th>the</th>
<th>sun</th>
<th>in</th>
<th>blue</th>
<th>amber</th>
<th>sky</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector for “The sun in the amber sky”</th>
<th>the</th>
<th>sun</th>
<th>in</th>
<th>blue</th>
<th>amber</th>
<th>sky</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Frequency magnitudes representations of strings in (10):**

<table>
<thead>
<tr>
<th>Vector for “The sun in the blue sky”</th>
<th>the</th>
<th>sun</th>
<th>in</th>
<th>blue</th>
<th>amber</th>
<th>sky</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector for “The sun in the amber sky”</th>
<th>the</th>
<th>sun</th>
<th>in</th>
<th>blue</th>
<th>amber</th>
<th>sky</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With contexts represented as such, the K-Means algorithm can then be instructed to sort the data into k clusters. Given that similar senses of a polysemous word will tend to have similar contexts, i.e., similar vectors, grouping should occur according to word sense.
3.7 Dimensionality Reduction

The EM and the K-Means algorithms function by manipulating the nuances of types in the data. A larger lexicon of types can significantly slow the processing time of the algorithms. Furthermore, there will tend to be many types in the data that occur either so frequently or so infrequently in contexts that they do not contribute to, and may in fact detract from, the effectiveness of the WSI.

Consider the scenario where a type, perhaps the word *the*, occurs in almost every context in the corpora. While it is a valid argument that *the* may be more likely alongside one sense of a polysemous word than another (and so be useful in WSI), due to its general omnipresence, its $P(\text{the}|\theta_M)$ values are likely to be inconsequential in most classification cases.

Often a *stop list*, a set of words to be ignored during processing, will be used in WSD and WSI (and other NLP) tasks. A stop list tends to consist of ‘semantically empty’ and frequently occurring words for the language being processed. A sample stop list for English, found online at ranks.nl (Ranks.nl, 2015) begins: {a, about, above, after, again, against, all, an, an, and, any, are, ... }.

Allow us to visualize the effect of applying this form of dimensionality reduction. The first of the two synthetic plots below illustrates frequently occurring words in the context of the polysemous word *rock*. The next illustrates the same after the application of a stop list.
A great deal of the ‘head’, the leftmost part of the frequency curve, is reduced by this application, allowing for faster and more accurate WSI processing. However, in the spirit of operating on entirely raw data, a stop list was not employed as a method of dimensionality reduction in this project. The process of deleting words’ morphological affixes is known as ‘stemming’. Stemming algorithms, or stemmers, can be very useful in dimensionality reduction, as they often group terms that differ slightly lexically, but are extremely similar semantically.

<table>
<thead>
<tr>
<th>Before Stemming</th>
<th>After Stemming</th>
</tr>
</thead>
<tbody>
<tr>
<td>refining</td>
<td>refine</td>
</tr>
<tr>
<td>refined</td>
<td>refine</td>
</tr>
<tr>
<td>refine</td>
<td>refine</td>
</tr>
<tr>
<td>unaffected</td>
<td>affected</td>
</tr>
</tbody>
</table>

Considering the various affixes that occur in English alone; plurals, gerunds, adjectival forms, genitives, and so on; it is easy to see how substantially the dimensionality of data can be refined by this process. Unlike applying a stop list, stemming does not discard information. Rather, it reorganizes it. If the ‘before stemming’ column in (11) were represented as a bag of words vector with frequency for dimensional magnitudes, it would have three dimensions and the magnitudes of its dimensions would sum to three. If the ‘after stemming’ column in (11) were represented as the same type of vector, it would have only two dimensions, but its dimensional magnitudes would still sum to three.

It is easy to imagine how a WSI program that does not think of refining and refine as of entirely separate entities will perform better than a WSI program that cannot amalgamate their frequencies or unite their occurrences. However, as with the stop list, the stemmer was seen to be unfaithful to the goal of unsupervised word sense induction due to its extensive prior knowledge of language.
3.8 Attenuated Vector Space

As a means of performing dimensionality reduction with fewer prerequisite linguistic notions, a gently assumptive model was developed. Sighting an application in the vein of stemming, methods of comparing vector representations of words were explored.

There are a number of models for comparing the difference between strings. The Levenshtein distance is one such model. It compares two strings by attempting to change one into the other. Levenshtein “allows insertions, deletions and replacements” (Navarro, 2001) of characters, assigning costs to each operation. The idea is that the more costly it is to make the strings equal, the less similar they are.

As such, it seems viable that a type of pseudo-stemming could be performed by comparing two strings and changing all instances of the longer of the pair into the shorter if the edit cost is below a certain threshold.

For example, if the cost of transforming *lanced* into *lance* is $x$, and $x$ is less than the threshold cost $t$, then instances of the *lanced* in the corpora could be replaced by *lance*. However, by Levenshtein, the cost of transforming *lanced* into *lance* is precisely the same as the cost of transforming *glance* into *lance*.

This latter example shows that Levenshtein comparison is not an adequate replacement for stemming, as *glance* and *lance* are semantically very different words.

In much the same way that the words in a string may be used to represent a string as a vector, the letters of a word may be used to represent a word as a vector.

(12)

```
a b c d e f g h i j k l m n o p q r
1 0 0 1 1 0 0 0 3 0 0 1 1 2 1 0 0 0
s t u v w y x z
1 1 0 0 0 1 0 0
```

In (12), the word *dimensionality* is represented as a vector. Possible alphabet types make up its dimensions. The magnitude of each dimension is equal to the number of tokens of that dimension’s character in the word.

Represented in such a way, the cosine similarity or Euclidean distance between two word vectors may be calculated. These are common means for deriving the likeness of two items. In the former, “the angle between two vectors is used as a measure of divergence between the vectors, and cosine of the angle is used as the numeric similarity (since cosine has the nice property that it is 1.0 for identical vectors and 0.0 for orthogonal vectors).” (Singhal, 2001). However, any cosine similarity or Euclidean distance measure taken as such also falls victim to this vector representation’s dismissal of letter order.

Representing a word as a vector whose dimensions are, as before, alphabet types, relevance may be given to letter order by setting dimensions’ magnitudes non-uniformly when the word is parsed.

An example will be given for a language that is read from left to right, and that tends to modify its stems with suffixes. While this is still incongruent with the aspiration to perform WSI on truly raw data, much less prior information is required than is by a stemmer (which works with extensive
morphological knowledge). Different settings would allow this model to work for languages that are read from right to left, or that tend to use different affixal arrangements.

For a language that is read from left to right and that tends to use suffixes to modify its stems, words can be represented in vector space for letter order sensitive comparison as follows:

(A) Inspect the corpora to find all possible alphabet types and let them represent the dimensions of each vector in step (B).

(B) For every unique word in the corpora, initialize a vector at the origin.

(B) (i) Let x equal some constant, a. Read the leftmost character of the word. Assign the corresponding vector dimension the value x.

(B) (ii) Subtract a fraction, f, of x from x.

(B) (iii) If there is another character in the word, add the magnitude x to its corresponding vector dimension, then go to step (ii).

Representing words as vectors in this way affects heavily weighted early characters and lightly weighted later characters. If two attenuated word vectors are contrasted, by calculating their Euclidean distance or cosine similarity, those differing in their first few characters will be found less similar than those differing in their last few characters. For example, lance will be closer to lanced than it is to glanced.

The figure on the left below shows a regular vector representation of the word egg: The figure on the right shows an attenuated vector representation of egg, where \( a = 1 \) and \( f = \frac{1}{4} \). Dimensions are represented on the x-axis and dimensional magnitudes on the y.

By comparing every word in the corporal lexica to every other word in the corporal lexica, finding the Euclidean distance between the attenuated vector representations of the words, and replacing the longer word with the shorter word if they are found to be below a certain distance threshold, a form of unsupervised pseudo-stemming is achieved. The process is repeated until there are no changes left to be made.

Below is some illustrative output that was generated by applying the attenuated vector distance comparison to the corpora of contexts of the
The underlined conversions are ones that are considered to be erroneous. For example, *form* should not be changed for *for*. However, not all such errors are necessarily harmful to WSI or WSD. Given that *they*, *then* and *the* are all likely to be found on a stop list, amalgamating their counts will not necessarily affect context classification. Furthermore, this may be useful in head and tail trimming, as will be explained in the following paragraphs.

The two parameters described above, a and f, may be modified for different word lengths, avoiding situations where long, initially similar and later very different words are found as non-distant. The results above were achieved with a and f settings for three word lengths. For words less than 5 characters long, a=4 and f=2. For words between 4 and 10 characters in length, a=10 and f=2.2. For longer words, a=10, f=4. Furthermore, words of length 2 or less were not to be modified.

A way of finding useful settings for these parameters was briefly explored by running the dimensionality reduction process multiple times with different a and f values for different word limits while trying to minimize dictionary matches and, simultaneously, lexical substitutions.

Head and tail removal (the deletion of the leftmost and rightmost slopes of a frequency curve) could then be crudely, but effectively, carried out on the data simply by removing words occurring with a frequency of more than x or less than y. Here, x and y could be given values related to lexical frequency counts after other dimensionality reduction.

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Original</th>
<th>Modified</th>
<th>Original</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>disintegration → disintegrated</td>
<td>beds → bed</td>
<td>them → the</td>
<td>sand → sun</td>
<td>frosts → frost</td>
</tr>
<tr>
<td>seen → see</td>
<td>planta → clienta</td>
<td>amongst → among</td>
<td>plainly → plains</td>
<td>thickly → thick</td>
</tr>
<tr>
<td>summits → summit</td>
<td>them → them</td>
<td>sand → san</td>
<td>seen → see</td>
<td>yellow → yellow</td>
</tr>
<tr>
<td>agents → agent</td>
<td>rapid → rapid</td>
<td>they → the</td>
<td>greater → great</td>
<td>seen → see</td>
</tr>
<tr>
<td>thickness → thicker</td>
<td>evidently → evident</td>
<td>these → the</td>
<td>softened → softer</td>
<td>residual → residue</td>
</tr>
<tr>
<td>removed → remove</td>
<td>residual → residue</td>
<td>these → the</td>
<td>residual → residue</td>
<td>evidently → evident</td>
</tr>
<tr>
<td>shifted → shifts</td>
<td>evidently → evident</td>
<td>these → the</td>
<td>shifted → shifts</td>
<td>evidently → evident</td>
</tr>
<tr>
<td>evidently → evident</td>
<td>shifted → shifts</td>
<td>then → the</td>
<td>shifted → shifts</td>
<td>evidently → evident</td>
</tr>
<tr>
<td>flints → flint</td>
<td>contains → contain</td>
<td>these → the</td>
<td>materials → material</td>
<td>envelopes → envelope</td>
</tr>
<tr>
<td>produced → produce</td>
<td>produced → produce</td>
<td>these → the</td>
<td>produced → produce</td>
<td>produced → produce</td>
</tr>
<tr>
<td>exposures → exposed</td>
<td>usually → usual</td>
<td>clearly → clear</td>
<td>rapidly → rapid</td>
<td>removed → remove</td>
</tr>
<tr>
<td>frosts → frost</td>
<td>loosened → loose</td>
<td>plunges → plunge</td>
<td>blocks → block</td>
<td>effects → effect</td>
</tr>
<tr>
<td>steepened → steep</td>
<td>slopes → slope</td>
<td>blocks → block</td>
<td>they → the</td>
<td>seen → see</td>
</tr>
</tbody>
</table>

The underlined conversions are ones that are considered to be erroneous. For example, *form* should not be changed for *for*. However, not all such errors are necessarily harmful to WSI or WSD. Given that *they*, *then* and *the* are all likely to be found on a stop list, amalgamating their counts will not necessarily affect context classification. Furthermore, this may be useful in head and tail trimming, as will be explained in the following paragraphs.

The two parameters described above, a and f, may be modified for different word lengths, avoiding situations where long, initially similar and later very different words are found as non-distant. The results above were achieved with a and f settings for three word lengths. For words less than 5 characters long, a=4 and f=2. For words between 4 and 10 characters in length, a=10 and f=2.2. For longer words, a=10, f=4. Furthermore, words of length 2 or less were not to be modified.

A way of finding useful settings for these parameters was briefly explored by running the dimensionality reduction process multiple times with different a and f values for different word limits while trying to minimize dictionary matches and, simultaneously, lexical substitutions.

Head and tail removal (the deletion of the leftmost and rightmost slopes of a frequency curve) could then be crudely, but effectively, carried out on the data simply by removing words occurring with a frequency of more than x or less than y. Here, x and y could be given values related to lexical frequency counts after other dimensionality reduction.
4 Results

4.1 Parting Senses

Throughout the execution of the program, the order in which the contexts are parsed is preserved. For testing purposes, there was an equal number of contexts of rock in a musical sense and contexts of rock in a geological sense, totaling 354 contexts.

The EM and K-Means algorithms, the latter using binary vector dimensional magnitudes, were applied to data without dimensionality reduction and were instructed to distinguish two meanings. The result of this was an extremely erroneous classification. Only half of meaning A’s contexts were calculated to be meaning A, and meaning B showed similar inaccuracies.

Dimensionality reduction seemed requisite, and so the procedure was repeated, this time with a stop list (Ranks.nl, 2015) and a Python Natural Language Toolkit stemmer (Bird et al., 2009) preprocessing the context data. In figures (13) and (14), the highest word frequencies before and after this preprocessing are contrasted.
Using the stop list and NLTK stemmer, K-Means categorized 100% of meaning A as meaning A, and 47% of the meaning B contexts as meaning B. EM, however, did not perform any better than before.

The procedure was repeated, this time removing some of the head and some of the tail of the frequency curve. The K-Means showed similar results as the previous test. The EM, however, categorized 94% of meaning A contexts correctly, and 87% of meaning B contexts correctly.

The stop list and NLTK stemmer were then deactivated as preprocessing tools to see how the algorithms performed using only head and tail dimensionality reduction. Once more, the K-Means performed erroneously. EM, on the other hand, classified all of the meaning A contexts as meaning A, and 96.5% of the meaning B contexts as meaning B.

The attenuated vector comparison and substitution method proposed above was then used for preprocessing the data, along with head and tail frequency removal. The following figure is a plot of types vs their token frequencies in the data after this preprocessing:
Here, both EM and K-Means performed well. EM classified over 99% of meaning A as being of meaning A and 92% of meaning B as being of meaning B. K-Means classified over 99% of meaning A as being of meaning A and 84% of meaning B as being of meaning B.

When evaluating a classifier, the term recall refers to “the proportion of Real Positive cases that are correctly Predicted Positive”, and the term precision “denotes the proportion of Predicted Positive cases that are correctly Real Positives.” (Powers, 2007).

Concretely, precision for meaning A classification is equal to:

\[
\frac{\text{contexts correctly classified as meaning A}}{\text{contexts classified as meaning A}}
\]

EM: 176/189
K-Means: 176/203

And recall for meaning A classification is equal to:

\[
\frac{\text{contexts correctly classified as meaning A}}{\text{meaning A contexts}}
\]

EM: 176 / 177
K-Means: 176/177

Thus, this adaption of the program performed word sense induction with a best precision of 93% and recall of 99% for meaning A and best precision of 99% and recall of 93% for meaning B.

In the interest of verifying whether these methods would work on another language, the program was used to perform WSI on two French corpora of contexts occurring with the word rouge. The first corpus was developed from a text on communism (where rouge may be used to refer to the doctrine), the second from a text on colour theory. (See section: Resources).

Results were slightly less accurate than those performed on the English corpus. K-Means correctly categorized meaning A 90% of the time, but correctly categorized meaning B only 70% of the time. EM performed with random-seeming categorization. However, the attenuated vector parameters here were not modified for French (using a method like the one described in 3.8). It is imagined that doing so would improve results.

In general, over the course of experimentation, it was found that EM performs better with a more aggressive head and tail frequency removal, i.e., a narrow body of frequencies. Binary K-Means seems to work well with a large tail removal, and smaller context windows. Stemming using the NLTK stemmer seemed just as efficient as the attenuated vector method.

4.2 Seeing Change

With the system able to return convincing WSI results, it was ready to be applied to corpora that exhibited a word’s semantic change. As in Dynamic EM in Neologism Evolution (Emms, 2013), diachronous usages and contexts of the word bricked were retrieved using google’s date-specific search feature. Bricked, in recent years, has come to be used metaphorically to describe an electronic device’s defunction. This is distinguished from bricked in a constructional sense.
The corpora, timestamped for the years between 2000 and 2012, were analysed using the above outlined WSI system. The results follow. Here, Meaning A is the novel sense of the word.

<table>
<thead>
<tr>
<th>Year</th>
<th>Number of Contexts</th>
<th>Number Categorized as Meaning A</th>
<th>Number Categorized as Meaning B</th>
<th>% Contexts Categorized as Meaning A</th>
<th>% Contexts Categorized as Meaning B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>88</td>
<td>29</td>
<td>59</td>
<td>32.95</td>
<td>67.05</td>
</tr>
<tr>
<td>2002</td>
<td>95</td>
<td>33</td>
<td>62</td>
<td>34.74</td>
<td>65.26</td>
</tr>
<tr>
<td>2003</td>
<td>95</td>
<td>47</td>
<td>68</td>
<td>49.74</td>
<td>50.53</td>
</tr>
<tr>
<td>2004</td>
<td>89</td>
<td>28</td>
<td>61</td>
<td>31.46</td>
<td>68.54</td>
</tr>
<tr>
<td>2005</td>
<td>88</td>
<td>50</td>
<td>38</td>
<td>56.82</td>
<td>43.178</td>
</tr>
<tr>
<td>2006</td>
<td>64</td>
<td>46</td>
<td>18</td>
<td>71.88</td>
<td>28.13</td>
</tr>
<tr>
<td>2007</td>
<td>58</td>
<td>47</td>
<td>11</td>
<td>81.03</td>
<td>18.97</td>
</tr>
<tr>
<td>2008</td>
<td>57</td>
<td>51</td>
<td>6</td>
<td>89.47</td>
<td>10.53</td>
</tr>
<tr>
<td>2009</td>
<td>50</td>
<td>44</td>
<td>6</td>
<td>88.00</td>
<td>12.00</td>
</tr>
<tr>
<td>2010</td>
<td>58</td>
<td>56</td>
<td>2</td>
<td>96.55</td>
<td>3.45</td>
</tr>
<tr>
<td>2011</td>
<td>55</td>
<td>47</td>
<td>8</td>
<td>85.45</td>
<td>14.55</td>
</tr>
</tbody>
</table>
The chart above, for each year examined, plots the percentage of contexts taken to be meaning A and the percentage of contexts taken to be meaning B. The latter is represented by the thicker line. A clear increase in meaning A’s, the neologism’s, categorization is seen. This is congruent with what one would expect to observe as one word sense gains popularity over time. Monitoring changes in these categorization values over time, and raising some sort of flag when values change dramatically (as is seen above between 2004 and 2006), a system could be automated to detect semantic change. If numerous intermittent timestamps and corpora were available, reacting to a heightened rate of change of these curves could give rise to a means of predicting new slang.
5 A Word on Coding

The Python language was predominantly used to write this project. Its renowned simple and clean appearance should facilitate following the code in appendices A and B for anyone who wishes to do so. Furthermore, well-established natural language processing toolkits (Bird et al., 2009) are readily available for use in Python. This is another reason for its selection. The system consists of two main applications, both of which are described extensively via comments in the source. The first deals with harvesting contexts for a particular word from texts or tables. The second implements the word sense induction via EM, K-Means and various optional dimensionality reduction methods. As a means of displaying the results of the above two programs, sections of code wrote data to isolated files for interpretation by a locally hosted dynamic website. Common web languages HTML, CSS, PHP and JavaScript combined to display findings and processes visually. (Appendix C).

6 Discussion

The system developed and the tests carried out above attest to Warren's slitted mask theory and demonstrate how word sense induction can be performed on untagged data using unsupervised methods and subtle, highly uninvolved dimensionality reduction. In theory, this system could be used to actively analyse language for neologisms and other meaning change. This would, however, require large and constantly updating corpora (perhaps pulling text from various dynamic websites), and a reasonable amount of processor dedication; suspected polysemous or novel words must be tried one at a time. One obvious improvement on the system would be to incorporate Yarowsky’s one sense per discourse and one sense per collocation findings, weighting polysemous words’ contexts accordingly. Furthermore, the attenuated vector style of representing data could be applied to contexts for use with K-Means of similar vector comparisons in order to factor co-occurring words’ polysemous proximities into calculations. Much work has been done in the past where word contexts are represented using a vector space model. Relational similarity and attributional similarity between words or word pairs can be estimated by such means. “When two words have a high degree of attributional similarity, we call them synonyms. When two pairs of words have a high degree of relational similarity, we say that their relations are analogous. For example, the word pair mason:stone is analogous to the pair carpenter:wood.” (Turney, 2006).
Measuring the degree of attributional similarity between the same words in different texts could be a simple way of detecting semantic change. One’s intuition might be that the attributional similarity between instances of *bricked* in the early 2000s is higher than in the late 2000s.

EM’s \( P(w_n|M_x) \) values could also be contrasted for different senses as a means of estimating attributional similarity.

It would be interesting to analyse WSI data in an effort to discern the type of semantic change taking place. Amelioration and pejoration, for example, might by nature show smaller or differently distributed changes in probable context words than metaphorization or metonymization.

### 7 Conclusion

This project explored unsupervised word sense induction and its application to detecting semantic change. It also found polylingual use for a position sensitive vector comparison model. Programs written for this project perform word sense induction with a satisfactorily high success rate, in congruence with the literature explored.

Ideas about predicting semantic change, carrying out WSI by other means, and deriving new nuances from WSI results have also arisen.

At the same time, this essayist has finally managed to put to use a long ineffectual interest in etymology, and has compounded it with a set of tools learnt affably over the last four years.

From here, it is only hoped that a continued interest in the field will provide opportunity to explore it further, and that there will be still more to say.
8 References


9 Resources

The French text on Communism may be found here:
https://hal.inria.fr/file/index/docid/202426/filename/aunoblethese.pdf


The French text on Colour Theory may be found here:
[Accessed 15 March 2015].


An application for generating charts, Highcharts, was used to visualize some data output of this program. Some figures are reproduced from these outputs and synthetic material. This is licenced under the creative commons licence. (http://www.highcharts.com/products/highcharts)
Appendices

Non-essential parts of these appendices are omitted in order to save space. The full appendices and all code may be viewed on the compact disk accompanying this project.

Appendix A: Text and Table Parser

main.py

```python
import control, textFunctions, generateContextMatrix, readTable, re

while 1 is not 2:
    # Get the location of the text to be read
    docLocation = control.control()

    # Context size
    # I.e.: the number of words to the left and the right
    # of the word in focus that are to be examined
    print("Please input a context window size.")
    contextRange = input()

    # Read in the text file
    text = textFunctions.fileToString(docLocation)

    # Check whether text or csv
    isCSV = re.match(r'.*\csv', docLocation)
    if isCSV:
        print("This is a csv file. Would you like to (1) analyse all contexts in the entire document, or (2) each line for a particular word’s context? Enter (1/2)")
        response = input()
        if response == "1":
            # Put the cleaned csv into an array
            cleanedTextList = textFunctions.textStringToList(textFunctions.cleanText(textFunctions.csvToText(text)))
    elif response == "2":
        readTable.main(text)
        quit()
    else:
        print("Your input was not understood.")
        quit()

else:
    print("It is assumed that this is a .txt file. Enter 1 to create a table of contexts with a particular word. Enter 2 to generate a table of word co-occurrence frequencies.")
    response = input()
    if response == "1":
        # Put the cleaned text into an array
        cleanedTextList = textFunctions.textStringToList(textFunctions.cleanText(text))
        print("Which word’s contexts would you like to search for?")
        subjectWord = input()
        csvFile = generateContextMatrix.createContextsCSV(subjectWord, cleanedTextList, contextRange)
        print("Would you like to save this file? (y/n)")
        response = input()
        if response == "y":
            print("Enter a name for this file. The file will be saved as a .csv document.")
            response = input()
```

file = open(response+".csv", "w", encoding='utf-8')
file.write(csvFile)
file.close()
else:
    print("tchau!")

elif response == "2":
    # Put the cleaned text into an array
cleanedTextList = textFunctions.textStringToList(textFunctions.cleanText(text))
    # Put each different word form used into an array
textLexiconList = textFunctions.toLexiconList(cleanedTextList);
    # Generate a matrix with both the x and y axis listing the lexicon
    # and the body of the matrix indicating how many times a word in y
    # occurs within the context of x
    contextMatrix = generateContextMatrix.generateContextMatrix(cleanedTextList, textLexiconList, contextRange)
    didSave = control.save(contextMatrix)

readTable.py

import textFunctions, generateContextMatrix, control

def main(text):
    # Split the table into elements in a list
csvList = text.split("\t")
contextRange = 15
sampleList = []
sampleLexiconList = []
contextMatrixList = []

    # Only deal with the actual text of interest for each row
for i in range(1,len(csvList)):
    if i % 3 == 0:
        sampleList.append(csvList[i])

    # For each piece of text
for i in range(0,len(sampleList)):
    # Clean said text, storing each sample in a List
sampleList[i] = textFunctions.cleanText(sampleList[i])
    # Then break each of these cleaned texts into lists of words
sampleList[i] = textFunctions.textStringToList(sampleList[i])
    # Also get the lexicon lists for each of these
sampleLexiconList.append(textFunctions.toLexiconList(sampleList[i]))

print("Which word’s contexts are you interested in?")
response = input()
for i in range(0,len(sampleList)):
    if response in sampleList[i]:
        # Get the contexts for each word in these
contextMatrixList.append(generateContextMatrix.generateContextVector(response, sampleList[i], sampleLexiconList[i], contextRange))

    # Remove any lexicon words that did not occur with bricked, despite being in the potential lexicon
tempMatrix = []
for x in range(0,len(contextMatrixList)):
    tempMatrix.append([])
    for y in range(0,len(contextMatrixList[x])):
        if not contextMatrixList[x][y][1] == 0:
            tempMatrix[x].append(contextMatrixList[x][y])
contextMatrixList = tempMatrix

print("Would you like to save this document? \n(y/n/.join/..p) \nThe join command will amalgamate the context counts")
print("The ".p" (print) command will amalgamate the context counts, order them and print them to the console")
response = input()
if response == "no" or response == "n":
    print("tchau!")
elif response == ".join":
    control.saveVecJoinAs(contextMatrixList)
elif response == ".p":
    control.printVecJoinAs(contextMatrixList)
else:
    control.saveVecAs(contextMatrixList)

return


textFunctions.py

import re, codecs

def fileToString(filename):
    # All files must be encoded in utf-8
    thisFile = codecs.open(filename, encoding='utf-8')
    output = thisFile.read()
    return output

def cleanText(text):
    # Remove tags
    text = re.sub(r'<![^>]*>', '', text)
    # Delete apostrophes that are not surrounded by letters
    text = re.sub(r'(['\w'])', '', text)
    text = re.sub(r'('['\w']', '', text)
    # Replace any dash followed by whitespace, i.e., delete " - ", or "-this"
    text = re.sub(r'[------]['\w]', '', text)
    # Replace any dash following whitespace
    text = re.sub(r'['\w][------]', '', text)
    # Replace dashes with a space
    text = re.sub(r'[-\\-', '', text)
    # Replace anything that is not alphanumeric or a dash or an apostrophe
    text = re.sub(r'[^\w-]', '', text)
    # Delete two or more whitespaces
    text = re.sub(r'\s{2,}', '', text)
    # Delete whitespaces at the start or end of the string
    text = re.sub(r'^\w*', '', text)
    text = re.sub(r'\w\Z', '', text)
    # Text to lower case
    text = text.lower()
    return text

[etc...]
Appendix B: Word Sense Induction

```python
class ContextData:
    contexts = []
def __init__(self):
        self.contexts = []
def setContexts(self, listOfContexts):
    self.contexts = listOfContexts
def getgetContext(self, i):
    return self.contexts[i].getWordList()
def printContexts(self):
    for context in self.contexts:
        context.printContextWords()
def getContexts(self):
    allContexts = []
    for context in self.contexts:
        allContexts.append(context.getWordList())
    return allContexts
def numberOfTokens(self):
    output = 0
    for context in self.contexts:
        for word in context.contextWords:
            output += 1
    return output
def numberOfContexts(self):
    return len(self.contexts)
def parseList(self, listOfContexts):
    for table in listOfContexts:
        for context in table:
            c = Context()
            for tuple in context:
                for times in range(0,int(tuple[1])):
                    t = Word()
                    t.setWord(tuple[0])
                    c.pushContextWord(t)
            self.contexts.append(c)
def parseContextList(self, contextList):
    self.contexts = []
    for context in contextList:
        c = Context()
        for word in context:
            t = Word()
            t.setWord(word)
            c.pushContextWord(t)
        self.contexts.append(c)
def getTypes(self):
    # returns a list of all types in the corpus
    types = []
    for context in self.contexts:
        for word in context.contextWords:
            if not word.getWord() in types:
                types.append(word.getWord())
    return types
```

[etc...]

tools.py

```python
import codecs, random, stemmer, distance, classes

def fileToString(filename):
    # All files must be encoded in utf-8
    thisFile = codecs.open(filename, encoding='utf-8')
    output = thisFile.read()
    return output

def applyStoplist(contextData):
    f = fileToString("stoplist.txt")
    sl = f.split("\n")
    # In case the file is written in a different format (using carriage return)
    if len(sl) <= 0:
        sl = f.split("\n")

    cd = []
    cdIndex = 0
    for context in contextData:
        cd.append([])
        for word in context:
            if word[0] not in sl:
                cd[cdIndex].append(word)
            cdIndex += 1

    contextData = cd
    return contextData

# Loads a comma-seperated CSV file of two columns and x rows, where
def loadContexts(filename):
    document = fileToString(filename)
    docRows = document.split("\n")
    # Or, in case of different formatting:
    if len(docRows) <= 0:
        docRows = document.split("\n")
    for x in range(0,len(docRows)):
        docRows[x] = docRows[x].split(",")
    # In case there are there is one incomplete element at the end:
    if len(docRows[len(docRows)-1]) < 1:
        docRows.pop(len(docRows)-1)

    # We now have a list of the type [ [word],[freq in this context] ],[ [word],[freq in this context] ],
etc...
    listOfContexts = []
    contextIndex = -1
    for x in range(0,len(docRows)):
        if docRows[x][0] == "--start--":
            listOfContexts.append([])
            contextIndex += 1
        else:
            listOfContexts[contextIndex].append([docRows[x][0],docRows[x][1]])

    ##
    ## NOTE: NLTK stoplists may be applied for testing by uncommenting these lines ##
    ##
    # listOfContexts = applyStoplist(listOfContexts)
    # listOfContexts = stemmer.stem(listOfContexts)
```
return listOfContexts

[etc...]

---

import random, math, tools

...  

K-Means algorithm for use in word-sense disambiguation

# Overseer the execution of k-means grouping into 'meanings' groups
# The 'binary' variable, a boolean, changes the operating of this algorithm.
# When true, vectors record a type's occurrence. Otherwise, vectors record a
# type's frequency.
def kMeans(contextData, meanings, binary):
    # The vector has the same number of components as there are types
    types = contextData.getTypes()

    # Define the contexts as vectors
    contexts = contextData.getContexts()
    contextVectors = []
    for context in contexts:
        if not binary:
            contextVectors.append(getVec(context, types))
        else:
            contextVectors.append(getVecBinary(context, types))

    # Get some starting values for the centroids
    centroids = getNDistantVecs(meanings, contextVectors)

    # Create a list to log to which centroid group each context belongs
    contextGroups = []

    # Initialize the list with all contexts belonging to group number zero
    for x in range(0, len(contextData.getContexts())):
        contextGroups.append(0)

    # Perform the k-means algorithm
    newContextGroups = regroup(contextVectors, centroids)
    centroids = generateNewCentroids(contextVectors, newContextGroups, centroids)
    print("Starting re-grouping iterations")
    loopCounter = 0
    while contextGroups != newContextGroups and loopCounter < 100000:
        if loopCounter % 50 == 0:
            print(str(loopCounter) + " reorganizations made.")
        loopCounter += 1
        displayKMeans(contexts, contextGroups, newContextGroups)
        contextGroups = newContextGroups
        newContextGroups = regroup(contextVectors, centroids)
        centroids = generateNewCentroids(contextVectors, newContextGroups, centroids)
        print("\n\n--Final Groups--\n")

    print(newContextGroups)

    ''' Write newContextGroups to a file'''

    f = "../output/kres.txt"
def displayKMeans(contexts, contextGroups, newContextGroups):
    f = '../output/kmeansout.txt'
    file = open(f, "a", encoding='utf-8')

    for row in range(len(contexts)):
        if contextGroups[row] != newContextGroups[row]:
            file.write(\n            "\n            The context { "
            for word in contexts[row]:
                file.write(word + " ")
                file.write("} changed to group " + str(newContextGroups[row]))

    file.close()

    # Converts a context list into a vector of word occurrence counts
    def getVec(context, types):
        vec = []
        for t in types:
            vec.append(0)
        for word in context:
            i = types.index(word)
            vec[i] += 1
        return vec

    # Converts a context list into a vector of word occurrence booleans
    def getVecBinary(context, types):
        vec = []
        for t in types:
            vec.append(0)
        for word in context:
            i = types.index(word)
            vec[i] = 1
        return vec

    # Consider all the vectors in the corpus. Return the n indexes of those that are furthest apart
    def getNDistantVecs(n, contextVectors):
        if n < 2:
            return contextVectors[0]
        largestDist = 0
        largestIndices = [0,0]
        for x in range(0, len(contextVectors)):
            for y in range(0, len(contextVectors)):
                if euDistance(contextVectors[x], contextVectors[y]) > largestDist:
                    largestDist = euDistance(contextVectors[x], contextVectors[y])
                    largestIndices = [x,y]
        return largestIndices

        # Now we know the farthest apart two vectors

        file = open(f, "a", encoding='utf-8')
        printLineBreak = True
        for x in range(len(newContextGroups)):
            if x % 10 != 0:
                file.write(str(newContextGroups[x]) + ", ",")
            else:
                file.write("<br/>" + str(x) + "<br/>",)
            file.write(str(newContextGroups[x]) + "", "")
            printLineBreak = True
        file.close()
# If n is two, we can return these two vectors
if n == 2:
    return [contextVectors[largestIndicies[0]], contextVectors[largestIndicies[1]]]

if n == 3:
    # Prepare to return the two vcs from before and then their average too
    output = [contextVectors[largestIndicies[0]], contextVectors[largestIndicies[1]]]
    for x in range(2, n):
        output.append(getAverageVec(output))
    return output

if n > 3:
    # Prepare to return the two vcs from before and some appended arbitrary points
    output = [contextVectors[largestIndicies[0]], contextVectors[largestIndicies[1]]]
    for x in range(2, n):
        randVec = []
        for dimension in range(0, len(contextVectors[0])):
            r = random.uniform(0, 1)
            randVec.append(r)
        output.append(randVec)
    return output

def generateNewCentroids(contextVectors, newContextGroups, centroids):
    outputCentroids = []
    # For each group: 0, 1, (etc)
    for c in range(0, len(centroids)):
        holdGroupVecs = []
        # For every context vector
        for vec in range(0, len(contextVectors)):
            # If the vector is a member of the group
            if newContextGroups[vec] == c:
                holdGroupVecs.append(contextVectors[vec])
        # Use all the members of a particular group to re-assign the vector
        # of the first centroid
        outputCentroids.append(getAverageVec(holdGroupVecs))
    return outputCentroids

# Returns the average (mean) vector of the list of vectors passed
def getAverageVec(vecs):
    if len(vecs) <= 1:
        return vecs[0]
    componentSums = []
    # Initialize component sum with the correct number of zeros
    for vec in vecs[0]:
        componentSums.append(0)
    # For each vector
    for vec in vecs:
        # For each component
        for c in range(0, len(vec)):
            componentSums[c] += vec[c]
        # Get averages of componentSums values (dividing them by the number of vecs)
    for s in range(0, len(componentSums)):
        componentSums[s] = componentSums[s] / len(vecs)
    return componentSums

# Categorizes vectors with regards to which centroid they are nearest
def regroup(contextVectors, centroids):
    newGroups = []
    # For every vector
    for v in range(0, len(contextVectors)):
        distanceFromCentroids = []
        # Calculate its euclidean distance from each centroid

for c in range(0, len(centroids)):
    distanceFromCentroids.append(euDistance(centroids[c], contextVectors[v]))
newGroups.append(closestCentroid(distanceFromCentroids))
return newGroups

# Returns the smallest value from a list
def closestCentroid(distanceFromCentroids):
    closest = distanceFromCentroids[0]
    index = 0
    for x in range(0, len(distanceFromCentroids)):
        if distanceFromCentroids[x] <= closest:
            closest = distanceFromCentroids[x]
            index = x
    return index

# Returns the euclidean distance between two vectors of the same dimensionality
def euDistance(centroid, vector):
    # Subtract the vectors components from the relative centroid components
    componentsSubtracted = []
    for component in range(0, len(centroid)):
        componentsSubtracted.append(centroid[component] - vector[component])
    # Sum the squares of the results
    sum = 0
    for component in componentsSubtracted:
        sum += (component * component)
    result = math.sqrt(sum)
    return result

em.py

import tools

# Given two lists of word context results, this function returns, for every word in the alphabet of the first
# list, a ratio of its frequency v.s. its frequency in the second list
def ratiosRelativeTo(l1, l2):
    outputList = []
    alphabet = tools.getLexicon(l1)
    for word in alphabet:
        freq1 = 0
        freq2 = 0
        for line in l1:
            freq1 += line.count(word)
        for line in l2:
            freq2 += line.count(word)
        if(freq2 > 0):
            outputList.append([word, freq1/freq2])
        else:
            outputList.append([word, 0])
    return outputList

# Given a list of word context results, this function returns the probability of the token appearing in the list
def probsWordsInWordContextList(l1):
    outputList = []
    totalNumWordsInList = 0
    for line in l1:
        totalNumWordsInList += len(line)
    return outputList
for line in l1:
    for word in line:
        totalNumWordsInList += 1

alphabet = tools.getLexicon(l1)

for word in alphabet:
    freqWord = 0
    for line in l1:
        freqWord += line.count(word)
    outputList.append([word, freqWord/totalNumWordsInList])

return outputList

# Returns the max number of context words that should be parsed, given the tables that we're considering

def numberContextWords(table):
    count = 0
    for row in table:
        count += int(row[1])
    return count

# Takes a list in the form:
# contextData[x][y][z][θ][1]
# [x] -> list of context data
# [y] -> list of tables (contexts of a particular instance of the word in question)
# [z] -> list of rows
# [θ] -> list of meanings
# tuple in each row: [0] -> word, [1] -> frequency

def hiddenMeaning(contextData, numberOfMeanings):
    # Create a list of the types occurring in corpus
    types = contextData.getTypes()

    # Generate a list of each type and its probability in the corpus
    corpusWordProbs = contextData.getWordProbs()

    # Write the probability of each word type in each dataset to a file
    tools.writeList(corpusWordProbs, "probWordsFromEachTable.csv")

    wordGivenMeaning = []
    thetaMeaning = []

    # Set theta values equally
    for x in range(0, numberOfMeanings):
        thetaMeaning.append(1/numberOfMeanings)

    # Randomly set starting word given meaning values
    for m in range(0, numberOfMeanings):
        wordGivenMeaning.append([])
        arbitrary = tools.xArbitraryValuesSumToOne(len(types))
        for t in range(0, len(types)):
            wordGivenMeaning[m].append([types[t], arbitrary[t]])

    for x in range(0, 50):
        emResultList = em(contextData, types, wordGivenMeaning, thetaMeaning)
        wordGivenMeaning = emResultList[θ]
        thetaMeaning = emResultList[1]
        #print("The probabilities of each context word given meaning 1 are: ")
        #tools.printList(emResultList[0][θ])
        #print("The probabilities of each context word given meaning 2 are: ")
        #tools.printList(emResultList[0][1])
print("Theta values are: ")
print(thetaMeaning)

return [wordGivenMeaning, thetaMeaning]

def em(contextData, types, wordGivenMeaning, thetaMeaning):
    print("---------------- Iteration ----------------")
    # Compute the conditional probabilities of the hidden variable given the data
    conditProbMeaning = []
    probMeaningGivenData = []
    # For each meaning
    for meaning in range(0,len(thetaMeaning)):
        conditProbMeaning.append([])
        probMeaningGivenData.append([])
        # For each row of context data
        for contextRow in range(0,contextData.numberOfContexts()):
            # Multiply this theta value by each word given meaning value in the row
            probMeaningGivenData[meaning].append(thetaMeaning[meaning])
            rowContextList = contextData.getContext(contextRow)
            for token in rowContextList:
                for tuple in wordGivenMeaning[meaning]:
                    if token == tuple[0]:
                        probMeaningGivenData[meaning][contextRow] *= tuple[1]

    sumRows = []
    for contextRow in range(0,contextData.numberOfContexts()):
        sumRows.append(0)
        for m in range(0,len(thetaMeaning)):
            for m in range(0,len(thetaMeaning)):
                sumRows[contextRow] += probMeaningGivenData[m][contextRow]
        for m in range(0,len(thetaMeaning)):
            conditProbMeaning[m].append(probMeaningGivenData[m][contextRow]/sumRows[contextRow])

    columnConditProbSums = []
    for m in range(0,len(thetaMeaning)):
        columnConditProbSums.append(0)
        for contextRow in range(0,contextData.numberOfContexts()):
            columnConditProbSums[m] += conditProbMeaning[m][contextRow]

    # expectedCountList => [ [ [type1,val],[type2,val] ], [ [type1,val],[type2,val] ] ]
    expectedCountList = []
    # Instantiate expected count list with zeros
    for m in range(0,len(thetaMeaning)):
        expectedCountList.append([])
        for t in range(0,len(types)):
            expectedCountList[m].append([types[t],0])

    # For each possible meaning
    for m in range(0,len(thetaMeaning)):
        # For each possible type
        for t in range(0,len(types)):
            # For each context
            for c in range(0,contextData.numberOfContexts()):
                # For each item in that row
                contextHolder = contextData.getContext(c)
                for i in range(0, len(contextHolder)):
                    # Count the number of this type and multiply that by this rows conditional probability
                    if contextHolder[i] == types[t]:
                        expectedCountList[m][t][1] += conditProbMeaning[m][c]

    # Sum expected counts for each meaning


```python
def euDist(v1, v2):
    # If any one of the vectors is at the origin, return an absurdly large value
    # Do this because some vectors, such as ones containing numbers, are deemed
    # unsuitable for this type of comparison, and so are better left as distinct
    if vecAtOrigin(v1) or vecAtOrigin(v2):
        return 1000
    # Subtract the v2 components from the relative v1 components
    componentsSubtracted = []
    for component in range(0, len(v1)):
        componentsSubtracted.append(v1[component] - v2[component])
    # Sum the squares of the results
    sum = 0
    for component in componentsSubtracted:
        sum += (component * component)
    result = math.sqrt(sum)
    return result

def wordToVector(w):
    alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '']
    vec = []
    for x in range(0, len(alphabet)):
        vec.append(0)
    for letter in w:
        if letter in alphabet:
            index = alphabet.index(letter)
            vec[index] += 1
    return vec
```
def unsupStem(listOfContexts, lex, thresholdLen):
    replaceable = []
    replacements = []
    for word in lex:
        for compare in lex:
            if compare not in replaceable and compare != word and len(compare) > thresholdLen:
                if euDist(wordToAttenuatedVector(word),wordToAttenuatedVector(compare)) < 1:
                    if len(word) < len(compare):
                        replacements.append([compare,word])
                        replaceable.append(compare)

    changesMade = ["none"]
    # Make the changes to the actual listOfContexts
    
    # Delete this after presentation:
    file = open("./output/unsup.txt", "w", encoding='utf-8')
    loopCounter = 0
    while len(changesMade) > 0 and loopCounter < 100000:
        changesMade = []
        loopCounter += 1
        if loopCounter % 1000 == 0:
            if "none" not in changesMade:
                changesMade = []
        
        for context in range(len(listOfContexts)):
            for word in range(len(listOfContexts[context])):
                if listOfContexts[context][word] in replaceable:
                    index = replaceable.index(listOfContexts[context][word])
                    if index < len(compare):
                        file.write("changed " + listOfContexts[context][word] + " for " + replacements[index][1] + "\n")
                        changesMade.append(listOfContexts[context][word])
                        listOfContexts[context][word] = replacements[index][1]
            print("sets of substitutions made.")
        file.close()
        return listOfContexts

def levenshteinContexts(listOfContexts, lex, thresholdLev, thresholdLen):
    replaceable = []
    replacements = []
    for word in lex:
        for compare in lex:
            if compare not in replaceable and compare != word and len(compare) >= thresholdLen:
                if levenshtein(word,compare) < thresholdLev:
                    replacements.append([compare,word])
                    replaceable.append(compare)

    # Make the changes to the actual listOfContexts
    for context in range(len(listOfContexts)):
        for word in range(len(listOfContexts[context])):
            if listOfContexts[context][word] in replaceable:
                index = replaceable.index(listOfContexts[context][word])
                listOfContexts[context][word] = replacements[index][1]
    return listOfContexts

# Returns true if there are any digits in the string
def hasNumbers(inputString):
    return any(char.isdigit() for char in inputString)

[...]
# importance to those letters coming at the beginning of a word.
# This means that comparing two attenuated word vectors, the stem of the word (in a language where
# the stem comes to the left and reading is done from left to right) is more valuable in comparison
# than the root.

def wordToAttenuatedVector(w):
    alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', ' ']
    vec = []
    # If the string contains a number, return a vector at the origin ''
    if hasNumbers(w):
        for item in alphabet:
            vec.append(0)
    return vec
    # Otherwise, return the vector representation of it ''
    for x in range(0, len(alphabet)):
        vec.append(x)
    # Because this process behaves differently with different lengths of words, it's reasonable
    # to split words into categories based on their length
    if len(w) < 5:
        attenuationValue = 4
        for letter in w:
            if letter in alphabet:
                vec[alphabet.index(letter)] += attenuationValue
                # Decrease the attenuation value
                # attenuationValue = attenuationValue - (attenuationValue/(2*len(w))

    elif len(w) < 10:
        attenuationValue = 10
        for letter in w:
            if letter in alphabet:
                vec[alphabet.index(letter)] += attenuationValue
                # Decrease the attenuation value
                # attenuationValue = attenuationValue - (attenuationValue/(2*len(w))

    else:
        attenuationValue = 10
        for letter in w:
            if letter in alphabet:
                vec[alphabet.index(letter)] += attenuationValue
                # Decrease the attenuation value
                # attenuationValue = attenuationValue - (attenuationValue/(2*len(w))

    return vec

classifier.py

[etc...]
# Check which product is greater, and let the greater product's meaning be the classification of the word

```python
file = open("../output/afterEMProbs.txt", "a", encoding='utf-8')

meaning1WordProbTuples = tools.orderContextWordProbs(wordGivenMeaning[0])
meaning2WordProbTuples = tools.orderContextWordProbs(wordGivenMeaning[1])

meaning1WordProbTuples[0].reverse()
meaning2WordProbTuples[0].reverse()
meaning1WordProbTuples[1].reverse()
meaning2WordProbTuples[1].reverse()

for word in range(len(meaning1WordProbTuples[0])):
    file.write("<tr>")
    file.write("<td>" + meaning1WordProbTuples[0][word] + "</td>" + "<td>" + str(meaning1WordProbTuples[1][word]) + "</td>" + "<td>" + meaning2WordProbTuples[0][word] + "</td>" + "<td>" + str(meaning2WordProbTuples[1][word]) + "</td>")
    file.write("</tr>")

file.close()

allContexts = contextData.getcontexts()
classifiedStrings = []

# For each context data item
for context in range(0, len(allContexts)):
    stringTotals = []
    # Set string product totals with the respective theta value for the meaning being examined
    for m in range(0, len(thetaMeaning)):
        stringTotals.append(thetaMeaning[m])
        checked = []
        # For every word in the particular context being examined
        for word in allContexts[context]:
            if word not in checked:
                checked.append(word)
                # For every tuple of possible types in the meaning being examined
                for tuple in wordGivenMeaning[m]:
                    # If there is a match, multiply the word given meaning probability by the prior probability
                    if tuple[0] == word:
                        stringTotals[m] *= tuple[1]
        classifiedStrings.append([allContexts[context], whichMeaningIsMoreProbable(stringTotals)])
```

[etc...]

Appendix C: Accompanying Local Website

display.php

This is code is a part of the local website set up to display results of the WSI. The rest of the code for the website is included in the CD accompanying this project. This particular extract is used to display graphs.

```php
<?php
    $myfile = fopen("../.../Users/Alex/Documents/py/fyp/output/data.txt", "r") or die("Unable to open file!");

    // Load the graph data and split them up into any number of graphs
    $data = fread($myfile,filesize("../.../Users/Alex/Documents/py/fyp/output/data.txt"));
    $data = explode("ENDGRAPH", $data);
    $data = array_filter($data);

    [...]

    <script type="text/javascript">
        $(function () {
            $('#container').highcharts({
                title: {
                    text: '<?php echo json_encode($data[0][0]); ?>',
                    x: -20 //center
                },
                subtitle: {
                    text: '<?php echo json_encode($data[1][0]); ?>',
                    x: -20
                },
                xAxis: {
                    categories: '<?php echo json_encode($data[2]); ?>'
                },
                yAxis: {
                    title: {
                        text: '<?php echo json_encode($data[3]); ?>'
                    },
                    plotLines: [{
                        value: 0,
                        width: 1,
                        color: '#808080'
                    }]
                },
                tooltip: {
                    valueSuffix: ''
                },
                legend: {
                    layout: 'vertical',
                    align: 'right',
                    verticalAlign: 'middle',
                    borderColor: 0
                },
                series: ["...
```
echo 'data:';
echo $values2;

}?>
}};
}};

[etc...]