Date: 6 September 2020

**An assessment of two theses about authorship attribution**

DS7004 Coursework / Student No. 1720146

Chung Hoi CHIU

**Abstract:** In this coursework, two master theses that concern computational authorship attribution are assessed. The first thesis (Segarra, 2014) introduced a novel method called Word Adjacency Networks (WANs). In the thesis, WANs was also used to tackle certain long-existing and well-known disputes about early modern dramas. The second thesis (Gungor, 2018), is unpublished. ‘Unpublished’ nowadays usually just means having not been peer reviewed. In this thesis, a large corpus of nineteenth century texts was formed for use in writing the thesis. The corpus was also contributed to the UCI Machine Learning Repository for the public's use. Gungor's thesis used various traditional machine learning methods and the neural network method of word2vec to perform authorship attribution analysis. This coursework also gives brief and clear accounts on WANs and word2vec. This coursework and its related materials are deposited in <https://github.com/ericchchiu/u1720146_DS7004_courseworkCodeAndData> .

**1. Introduction**

Computational authorship attribution is an unconventional branch of the discipline of natural language processing. It uses texts of known authors to predict the author(s) of text(s) their author(s) are unknown. Its birth and progress are in parallel with the birth of computers and the improvement of computing power. Segarra's thesis (Segarra, 2014) introduced a novel authorship attribution method called Word Adjacency Networks (WANs). No code of this method has yet been disclosed. However, from the description of the method, it can be inferred that it consumes a great amount of computer power. Gungor's thesis (Gungor, 2018) formed a large digital corpus of nineteenth century texts. In addition to using traditional machine learning methods, the thesis used the neural network method of word2vec to perform analysis. Although with the help of the rapid growth of computing power and the rapid advancement of open sources, both methods can now be implemented with a laptop, they could not be implemented even with a mainframe ten years ago.

**2. Segarra's thesis**

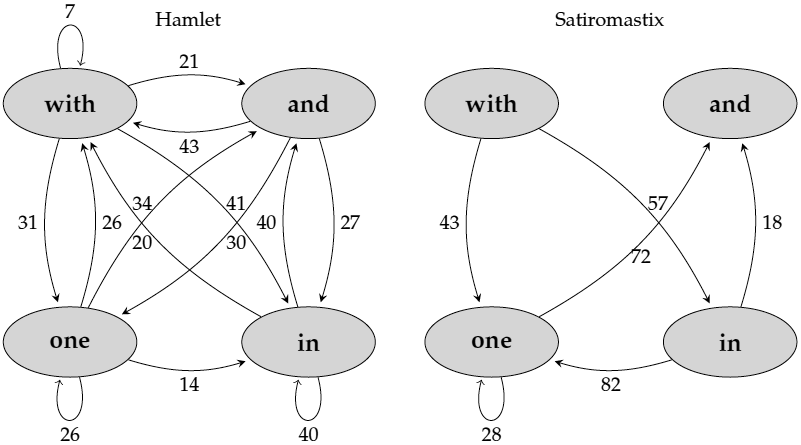
**2.1 A part or the whole of Segarra's thesis was published before and after completion of the thesis**

Segarra's thesis is dated 2014. However, before and after completion of the thesis, four peer reviewed papers covering a part or the whole of the thesis were published under the names of Segarra and people whom he thanked in his thesis. The co-authors include Mark Eisen, a scientist who specialises in machine learning and statistics; Alejandro Ribeiro, the supervisor of the thesis; and Gabriel Egan, a renowned Shakespearean (Segarra et al., 2013; Segarra et al., 2015; Segarra et al., 2016 and Eisen et al., 2018). The pre-published version of the 2015 and 2016 papers were already mentioned in Segarra's thesis and posted on the internet.

Segarra's thesis is about WANs, a novel computational authorship attribution method. We can infer from the four papers that Segarra, a candidate for a master degree, did not develop WANs alone, but that was developed by a team of experts who processed strong computing, machine learning and statistics skills and profound knowledge in early modern dramas.

**2.2 A very brief description of WANs**

The paper published by Shakespeare Quarterly (Segarra et al. 2016) provided a detailed and easy-to-understand explanation of how to build the two normalised WANs of words 'on', 'in', 'one' and 'with' (A WANs is in fact a Markov Chain in statistics):



(Fig. 2.1 Extracted from (Segarra et al. 2016, p. 238))

from these two pieces of early modern drama:

With one auspicious and one dropping eye,

With mirth in funeral and with dirge in marriage,

In equal scale weighing delight and dole.

(Shakespeare, Hamlet, 1.2.11–13)

I wonder then, that of five hundred, four,

Should all point with their fingers in one instant

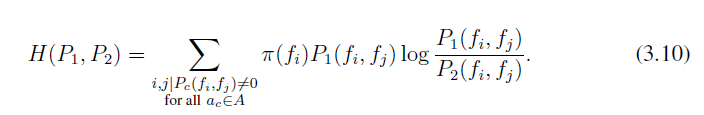
At one and the same man?

(Dekker, Satiromastix, 1.2.242–44)

(Requoted from (Segarra et al. 2016, p. 235))

(Note: In real situations, a text to be used to form a WANs would contain thousands of words, and the words chosen to construct a WANs would be dozens of selected function words, such as 'the',' to', 'I', 'your', etc.)

The information contained in the two WANs can then be plugged into the following formula to calculate their relative entropy:



(Segarra et al., 2014, p. 17)

In the above formula, P1 and P2 represent two WANs (Markov Chains), and H(P1, P2) represents the relative entropy between the two WANs. H(P1, P2) can have a value equal to or larger than zero. Zero relative entropy value means that the two WANs are identical. *i* and *j* represent two adjacent function words.

The footnote of p. 239 of Segarra et al., 2016 provided a detailed account of how to execute the above formula. However, a deeper explanation on the pi sign of the formula is helpful. It is called the 'markov chain limiting distribution', or 'limit probability'. To calculate the pi values, one should first represent a normalised WANs with a matrix. For example, we can represent the normalised Hamlet WANs as shown on Fig. 2.1 with the following matrix:

and in one with

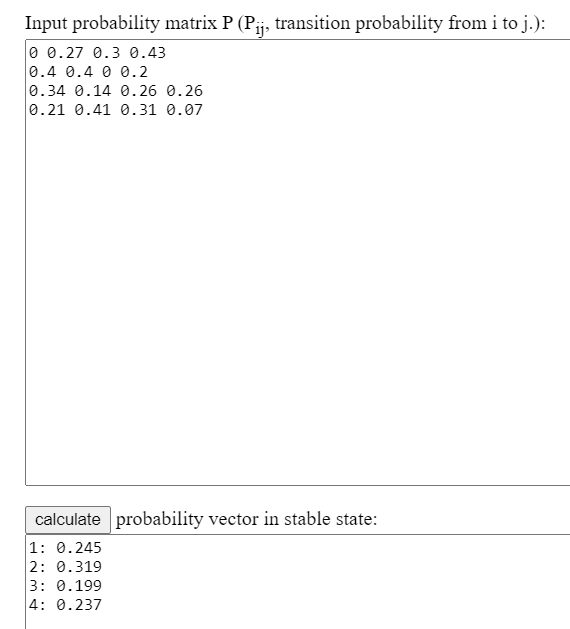
and 0 0.27 0.3 0.43

in 0.4 0.4 0 0.2

one 0.34 0.14 0.26 0.26

with 0.21 0.41 0.31 0.07

Then we can calculate the pi values by iterating a tedious calculation loop on the matrix until it reaches a stable state (i.e. no more significant change). One can easily do the calculation by using a free web calculator. Below is an image of a calculator which is located at <https://sites.google.com/view/kilin/software/finitemarkovchain/markov> ,with input (the above matrix) and output shown:



(Fig. 2.2)

Accordingly, for the normalised Hamlet WANs, pi(and) = 0.245, pi(in) = 0.319, pin(one) = 0.199 and pi(with) = 0.237

Even a matriculation standard student can understand how to produce a WANs and how to calculate the relative entropy by studying carefully the relevant pages of Segarra's thesis (about ten pages) (Segarra et al. 2014) and the paper published by Shakespeare Quarterly (about three pages) (Segarra et al. 2016).

Then, to do an authorship attribution exercise, one can:

1. form a WANs for every text of every candidate author and every text of the target texts finding their author is the aim of the exercise;
2. then average the edges of the WANs of each candidate author and the target texts to produce average WANs which are called profiles; and
3. calculate the relative entropy values between each profile and the profile of the target texts.

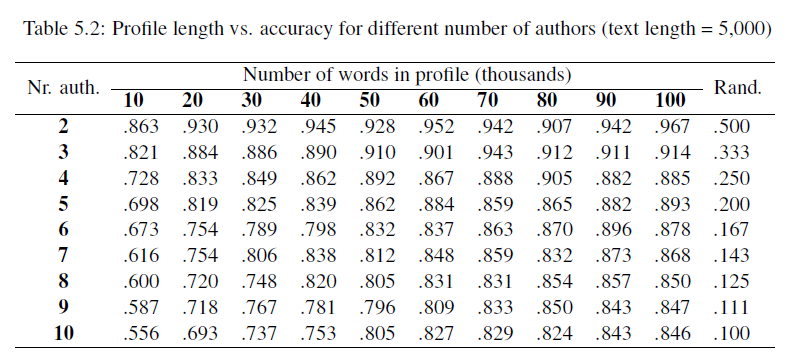
Then, the author whose profile produces the lowest relative entropy value is the author of the target texts as determined by WANs.

Although the procedures of constructing a WANs and finding a relative entropy value are not too difficult to understand, the mathematics behind the procedures (Kesidis and Walrand 1993) is not.

**2.3 Segarra's thesis rigorously proves that WANs outperforms other methods**

To prove that WANs method is powerful, Segarra's thesis produced a self-made nineteenth century novel corpus which consists of the works of 21 authors. Each author contributes an average of 560,000 words to the corpus. The thesis provides a url link to details about the corpus but the url is no longer valid.

Segarra's thesis used a well organised plan to test vigorously the performance of WANs. One of the results is shown below:



(Fig. 2.3, Extracted from Segarra et al. 2014, p. 29)

Fig. 2.3 shows that there are quite a few hyperparameters that need to be tuned. Such hyperparameters include:

1. the length of each text (here 5000);

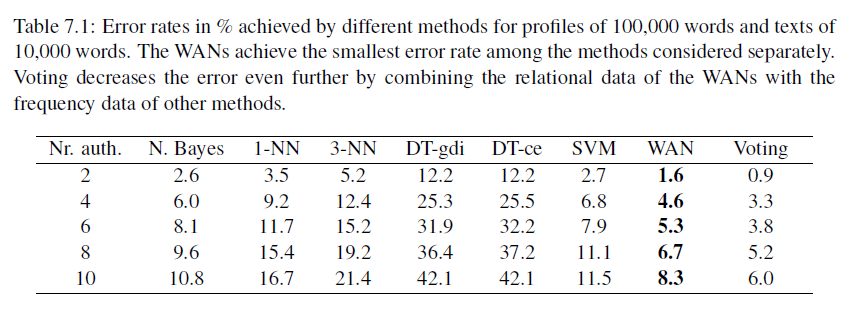
1. the number of total words of each candidate author to be used (here from 10,000 to 100,000); and
2. the number of candidate authors (here from 2 to 10).

In addition to the three hyperparameters, other hyperparameters that cannot be observed from Fig. 2.3 include:

1. the maximum distance (number of words) between two function words longer than it the two function words are no longer considered adjacent; and
2. the discount factor of the distances of function words (for example, if the discount factor is 0.8, then in the sentence 'but a swarm in July is not worth a fly', the distance between 'but' and 'a' is 1 and therefore there is no discount to the distance ( 10.8 = 1); however, the distance between 'but' and 'in' is 3 and therefore the distance should be discount to 2.4082 (30.8 = 2.4082)).

Segarra's thesis carefully, systematically and painstakingly tested WANs with various values of the hyperparameters and showed that, for using well-tuned hyperparameters, WANs can make very accurate attributions.

Then, finally, before leaving the nineteenth century novel corpus, Segarra's thesis provided a table which compared the performance of WANs with other machine learning methods. The table is shown below:



(Fig. 2.4, Extracted from Segarra et al. 2014, p. 48)

Fig. 2.4 showed that WANs outperforms all other machine learning methods. However, did Segarra also carefully tune the hyperparameters of the other methods?

**2.4 In Segarra's thesis, WANs is used to tackle long existing and well-known authorship disputes about early modern dramas**

In the second half of Segarra's thesis, WANs was used to study early modern dramas. A self-made corpus was formed which contained works of George Chapman, John Fletcher, Ben Jonson, Christopher Marlowe, Thomas Middleton and William Shakespeare. The texts were extracted from their carefully selected seventeenth century editions. Speech prefixes of the dramas were removed. Sentences were treated as ending at the end of speeches. Therefore, the issues emanated from punctuation carrying different meanings in the early modern era, and the special features of early modern dramas, such as enjambments (break lines) and stichomythias (split lines), no longer exist. The spelling variations in function words such as 'of' can be spelled as 'off', 'offe', or 'o', and 'with' can be spelled as 'wid', 'wyth', 'wytt', 'wi', 'wt' and 'wth', all of which were dealt with at the stage of forming the WANs, which means that the problem of spelling variations was dealt with programmatically, rather than by changing the texts. The above stated knowledgeable, careful and clever methods reflect the fact that the team that invented WANs consisted of data science experts and an early modern English expert.

Mainly because there was a Shakespearean in the team, in addition to the general studying of the early modern dramas, the WANs was also used to tackle long existing and well-known authorship disputes about early modern dramas, down to the act and scene level of certain dramas (a drama can be divided into acts, and acts into scenes). Hereinbelow is an example:

The act and scene analysis of Shakespeare and Fletcher’s other collaboration - Henry VIII - is displayed in Fig. 11.5. Recall that, when attributing the full play, Shakespeare was the top candidate while Fletcher was in fact ranked fourth, thus revealing no evidence of collaboration; see Fig. 9.5 or Fig. 11.2. We see similar results in Fig. 11.5, in which Shakespeare is assigned every act. Fletcher, again, is ranked poorly in every act. A scene by scene analysis between Shakespeare and Fletcher, however, does reveal Fletcher to be a stronger candidate than Shakespeare in several individual scenes. In fact, the scene breakdown we observe - in which Shakespeare is assigned scenes 1.1-2, 2.1-2, 2.4, 3.2, 4.1-2, and 5.1-2 and Fletcher is assigned scenes 1.3-4, 3.1, and 5.4, and 2.3 and 5.3 ties between both authors - is aligned to that proposed by Cyrus Hoy and currently accepted by many scholars. The primary area of disparity between the breakdown we propose and the one given by Hoy is the authorship of Act 4. While Hoy assigns Act 4 to Fletcher, we find that there is greater evidence that Shakespeare contributed this section. Both scenes are attributed to Shakespeare by a significant margin of at least 5cn. Another point of contention is the assignment of 2.3 - given to Shakespeare by Hoy - to Fletcher by a small margin.

(Segarra et al. 2014, pp. 84 - 85)

The long quotation above indicates that even a drama as a whole has already been attributed to an author by WANs and that even every act of the drama have also been attributed to the author, it may still be possible for WANs to attribute certain scenes underneath the acts to another author, which would support the proposition of a certain group of Shakespeareans. This approach is controversial.

As shown in Section 2.2 above (Fig. 2.1), even a 23-word text can form a WANs. It means that, after forming the profiles of Shakespeare and Fletcher (i.e. the average of their WANs) by using the texts as contained in the self-made early modern drama corpus, one can use the two profiles (WANs) to test whether a text was written by Shakespeare or by Fletcher, regardless of the length of the text. Furthermore, as discussed in 2.3 above, in the WANs method, so many hyperparameters are required to be tuned and different hyperparameter values will give different results. Therefore, if it is intended to use the WANs method to support an opinion on a well-known authorship dispute, the details of the application of the method such as the values of the hyperparameters, the length of the texts, etc., should be disclosed, or the experiments should be subjected to the scrutiny of an independent evaluator.

**3. Gungor's thesis**

**3.1 The nineteenth century text corpus**

The biggest achievement of Gungor's thesis is that it produced a large corpus of nineteenth century texts ( <http://archive.ics.uci.edu/ml/datasets/Victorian+Era+Authorship+Attribution> ). It consists of about 92,500,000 words written by 50 nineteenth century authors, the majority of them are novelists. I have painstakingly studied this corpus while writing the DS7003 coursework earlier this year. My opinions on this corpus and advice on how to use this corpus can be found from my coursework ( <https://github.com/ericchchiu/u1720146_DS7003_courseworkCodeAndData> ). Below are my additional opinions and advice:

1) Gungor uploaded an author list to his github repository (<https://github.com/agungor2/Authorship_Attribution/blob/master/author_list.txt> ) thirteen months ago. However, this list is still not correct.

2) The texts of the corpus were obtained from the huge database GDELT project. As stated in my DS7003 coursework, the quality of the texts obtained from GDELT is very poor. However, since Gungor was then only a student with limited resources, to form such a large corpus, he should have no alternative but to use this database. Below is a table that compare the numbers of books of three nineteenth century authors obtained by Gungor and available in two relatively easy to access corpora:

|  |  |  |  |
| --- | --- | --- | --- |
| Author | Gungor’s collection | Available in Gutenberg | Available in BL 19th C. Collection |
| George Eliot | 22 | 19 | 19 |
| Bret Harte | 64 | 25 | 19 |
| Catharine Maria Sedgwick | 18 | 1 | 3 |

(Note: Gutenberg is free to access, while the BL 19th Century Collection needs to be subscribed. The quality of texts in these two corpora is high.)

The table above indicates that, for famous authors such as George Eliot, the numbers of books available in the two corpora are similar to that collected by Gungor, for less famous authors, far fewer books were collected by the two corpora than were collected by Gungor.

3) Gugor did not provide labels to the test set of about 38,810,000 words. As all traditional machine learning methods for NLP are supervised learning methods, data without labels will be of no use to them. However, I guess that the test set is produced to form a word embedding model for use in a new kind of neural network methods called word embedding methods, such as word2vec, gloVe, etc. Labelling data is not required to form such a word embedding model. However, if a data set is intended to form a word embedding model, calling it a test set is misleading. Furthermore, it is still better to provide labels to the test set since people may use the test set for other purposes.

**3.2 Gungor did not pay attention to stop words**

I was very surprised when I came across the following passage when reading Gungor's thesis: 'At the early stage of our work, we have considered taking out all the stop words from the raw text data and keep (*sic*) the order of the rest of words.' (Gungor, 2018) Here 'early stage' means at the stage when traditional machine learning methods were used. It is a big mistake. Although in other branches of NLP such as sentiment analysis, stop words would usually be removed from texts before they are fed into a learning machine, the frequency of common words (most of them are stop words) is one of the two most successful features in authorship attribution studies (the other one is character n-gram). Removing stop words is the main reason why Gungor could not obtain satisfactory results when working on traditional machine learning methods (accuracies are about 40% to 80%).

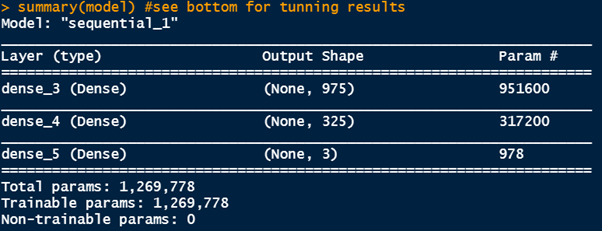
**3.3 Gungor did not pay attention to character n-gram**

Gungor's thesis touched on character n-gram, the most popular feature to be used for performing authorship attribution studies. However, he did not use the feature to perform his analysis.

**3.4 I produced a programme that uses character n-gram as the feature and neural network as the classifier to perform authorship attribution analysis**

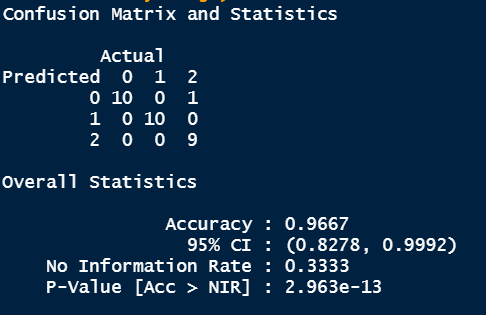
Gungor's mention of the character n-gram feature and his use of a new neural network method called word2vec (please see below) to perform analysis inspired me to write an R programme that uses character n-gram as the feature and the neural network as the classifier to analyse the same texts I used when doing the DS7003 coursework. The code is appended here (Appendix 1). Pictures of some of the outputs of the programme are shown below:

Structure of the neural network used (Two hidden layers which consist of 975 nodes and 325 nodes respectively, and a softmax layer):



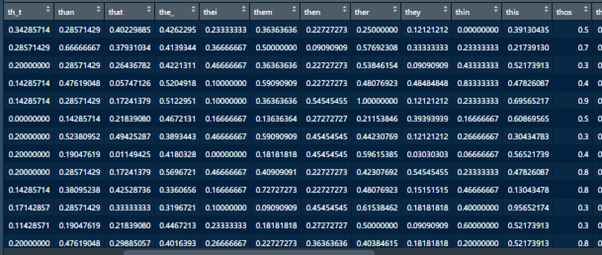
(Fig. 3.1)

The confusion matrix for the results obtained from using character 4-gram as the feature and the above neural network as the classifier:



(Fig. 3.2)

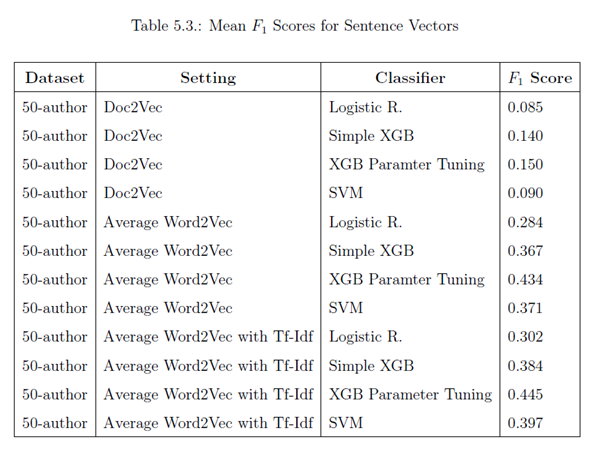
Part of the normalised character 4-gram table:



(Fig. 3.3)

**3.5 Gungor put a significant effort on word2vec**

Approximately 20% of Gungor's thesis concerned word2vec. However, the results obtained from those experiments using word2vec are not satisfactory (Please see Fig. 3.4 below), mainly because Gungor used all words of the text data to perform his analysis. However, in the field of authorship attribution, content words should be regarded as ‘noise’ and should be ignored.



(Fig. 3.4, extracted from Cungor, 2014, p. 71)

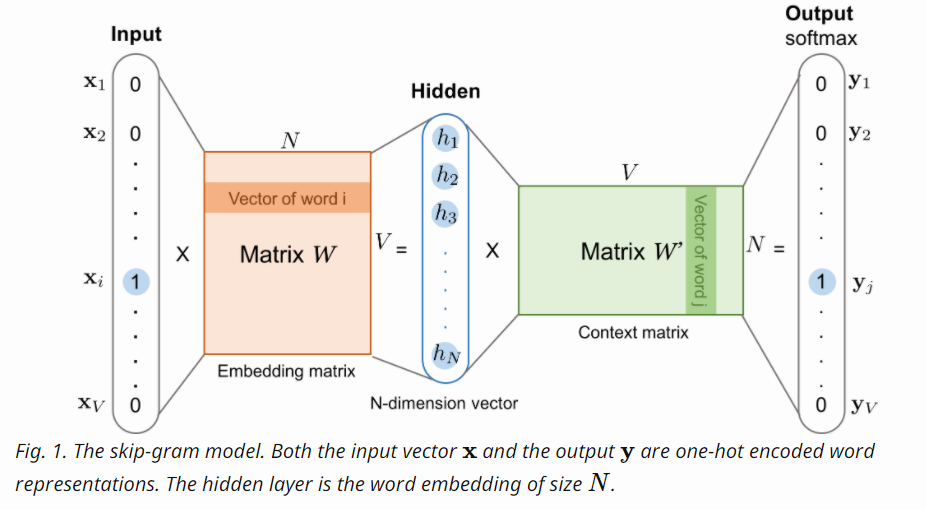
Although Gungor paid a significant amount of attention to word2vec, he did not explain what it is. Therefore, in the next subsections, I will provide a very brief explanation on what word2vec is. I will then provide two programmes to show how word2vec can be used to perform authorship attribution analysis.

**3.6 A brief explanation of word2vec**

I found that, when explaining word2vec, people usually focus on the special techniques (tricks) that it uses, such as skip-gram, continuous bag of words (CBOW), negative sampling and hierarchical softmax, but they ignore its architecture. To understand word2vec, understand its architecture is more important than understanding the tricks it uses. I will briefly explain the architecture of word2vec hereunder.

I will use the picture below obtained from Lilian Weng's blog (Weng, 2017) for my explanation:

1) word2vec is a specially arranged single hidden layer neural network



(Fig. 3.5, extracted from (Weng, 2017))

The neural network system for training a word2vec model (a vector space) is in fact a series of three matrices (actually, it should be two, see 2) b) below). They are described below:

a) The input matrix: if the total number of vocabularies of the input texts is V, then the size of the input matrix is V x V. Every row of the matrix represents one word. In a row, only one of the items is one; the other items are all zeros (this is called one-hot encoding technique). However, for the reason explained in 2) b) below, in real situations, this V x V one-hot encoding matrix need not be produced.

b) The embedding matrix: The hidden layer shown on Fig. 3.5 is in fact the representation of the columns of this matrix (or the rows of the context matrix as mentioned in c) below). The size of this embedding matrix is V x N.

c) The context matrix: It is a N x V matrix.

2) How a word2vec model operates:

a) Tiny random numbers should first sow to the embedding matrix (V x N) and the context matrix (N x V), and they collectively are called weights. (The total number of weights is 2 x V x N. They need to be tuned. If the vocabularies V is 100,000 and the number of nodes of the hidden layer N is decided to be 300, the number of weights that must be tuned is 60,000,000. (2 x 100000 x 300). Theoretically, we can withdraw the hidden layer and train a V x V matrix directly. If this is the case, the number of weights needed to be tuned will be 166.67 times more (100,000 x 100,000 = 10,000,000,000 and 10,000,000,000/ 60,000,000 = 166.67))

b) Then, after a word (for example, 'fox') is fed into the model, only the row of the input matrix that corresponds to the word 'fox' (1 x V) will be used to multiply the embedding matrix (V x N) to obtain a 1 x N row matrix. Since the 1 x V row is in fact a one-hot encoding row, the multiplication therefore just means picking up the row that represents 'fox' from the embedding matrix to form a 1 x N row matrix. Consequently, there is no need to form the V x V input matrix.

c) Then we multiply this row matrix (1 x N) with the context matrix (N x V) to obtain a 1 x V row matrix.

d) Then we use the softmax function to normalise the 1 x V row matrix to a probability distribution.

e) If the word 'fox' (the target word) is in fact in the sentence 'The quick brown fox jumps over the lazy dog' and we consider the two words on both sides of the target word the adjacent words of the target words, then the adjacent pairs will be {fox, quick}, {fox, brown}, {fox, jumps} and {fox, over}. Then we compare these true outcomes of adjacent words of 'fox' (i.e. 'quick', 'brown', 'jumps' and 'over', one at a time) with the probability distribution obtained from d) above, and propagate back the 'errors' for adjusting the tiny weights contained in the embedding matrix and the context matrix. As more texts are fed into the model, the weights will gradually be tuned to provide more accurate predictions of the adjacent words of the input word.

3) What word2vec wants is not the trained model:

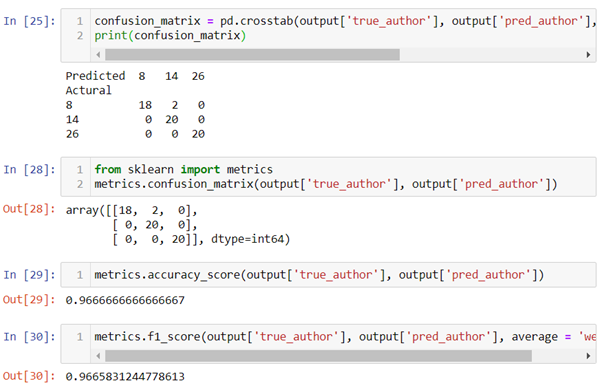
work2vec is not interested in predicting which word will most likely appear adjacent to the input word. The aim of work2vec is to obtain the trained V x N embedding matrix (it is called word embedding model). After training, only the word embedding model will be retained. This model is an N-dimensional vector space with V number of vectors in it. Each vector represents a vocabulary word of the input texts.

4) Words with similar meanings will be clustered together:

In a trained word embedding model, the vectors of words with similar meaning would be clustered together, which is a very valuable feature. Because it is able to produce such a word embedding model with such a feature, word2vec technique is now widely used in sentiment analysis, recommender system building, social bias detection, information retrieval, etc. Some interesting uses of word2vec are shown in Appendix 2.

**3.7 Using word2vec to perform authorship attribution analysis**

After understanding the architecture of word2vec, I adapted a Python programme to perform an authorship attribution exercise on texts (extracted from Gungor's corpus) of Charles Dickens (author number: 8), George Eliot (14) and Jane Austen (26). The programme is appended here as Appendix 3. The results of running the programme are shown below:



(Fig. 3.6)

The results are good (f1 = 0.97), mainly because in my programme, only vectors of stop words are used for calculation. This approach is exactly the opposite of what Gungor did (filtering out all stop words). However, the programme took approximately 45 minutes for my laptop to run. word2vec is just a specially arranged single hidden layer artificial neural network, and ANN is notoriously very time-consuming.

**3.8 I cannot understand one group of Gungor's experiments**

In Gungor's thesis, a few pages after the table of the not satisfactory results obtained from the application of word2vec (see Fig. 3.4 above), I found the passage below:

To implement this algorithm, we have used pre-trained Google vectors to calculate sentence vectors. To see the performance of this algorithm, text pieces are chosen from the works of W. Irving, F. H. Burnett, J. Abbott, J. Payn, O. Optic and there is no unknown author in the test data. In this setting, our implementation of Unsupervised feature learning has performed %92 accuracy by choosing window size as 8 and stride size as 4. When we also implement window size as 10, stride size as 5, and concatenating these two features have performed %97 accuracy. In the same experimentation setting, the bag of words accuracy has been recorded as %99.

Gungor's thesis also provided the url to the programme for doing this set of experiments which, according to the thesis, provided the miraculously good results of 92%, 97% and 99% respectively: https://github.com/agungor2/Authorship\_Attribution/blob/master/Unsupervised\_

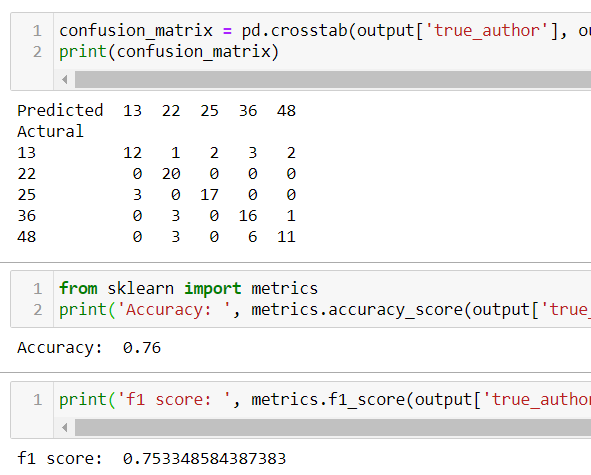
feature\_learning.m

I cannot understand the above quote and the programme. It is written in matlab language. Furthermore, I found most input data of this programme are related to stop words (Vocabulary\_wstopwords.mat, ml\_challenge\_data\_wstopwords.mat and word2vec\_data\_stop\_words.csv), but Gungor did not explain what they were for. Gungor did provide the following explanation on why this time the results were so good:

One of the main reasons why bag of words and unsupervised feature learning have performed well on the test data is because of the test and train data split. As noted before, when splitting train and test data if the book ids are not uniquely distributed then the classification task becomes an easy job.

I also cannot make sense of this passage. He should not be bothered with training data because, as mentioned in the previous quote, he 'used pre-trained Google vectors'.

It appears that the 'pre-trained Google vectors' that Gungor mentioned should be the famous 3.39GB pre-trained word2vec model GoogleNews-vectors-negative300. I downloaded this pre-trained model and wrote a programme (Appendix 4) to use it to perform authorship attribution studies on texts of the five authors as mentioned in the previous quote. The results of running the programme are shown below:



(Fig. 3.8)

The results are not satisfactory (f1 = 0.75). After careful studying, I found the main reason for the unsatisfactory results is that, for reasons that have not been published, the stop words 'to', 'hasnt', 'a', 'of', and 'and' were filtered out from GoogleNews-vectors-negative300. Then, I was even more surprised by Gungor's ability to achieve 92%, 97% and 99% accuracies respectively when he used the pre-trained model to perform three authorship attribution experiments.

**4. Conclusion and future work**

The WANs method introduced by Segarra's thesis is attractive because, unlike other authorship attribution methods that consider only frequencies of occurrence of certain features such as those of function words, WANs also considers permutation and distance between function words.

However, surprisingly, up to now, not a line of code or the programming language used for implementing the WANs method has been disclosed, not to mention the details of the experiments performed for tackling long-existing and well-known disputes. Therefore, at the moment, I would rather follow a Shakespearean's advice:

We should also note that practitioners of the Word Adjacency Networks method have, at the time of writing, yet to disclose their actual results. Readers, of course, should not accept authorship claims without seeing the actual results.

(Freebury-Jones, 2019)

Gungor contributed the large nineteenth century text corpus that he produced for writing his master thesis to the UCI Machine Learning Repository. It is the only corpus in the UCI that is expressly designated for authorship attribution studies. However, the quality of the texts there is bad, the list of authors provided is not correct, and the test set has no labels. Gungor also did not pay attention to stop words, as the result of which he was not able to obtain good results when using either traditional machine learning methods or the neural network method of word2vec for analysis.

However, Gungor's thesis inspired me to use stop words as the feature and word2vec as the classifier to do experiments on texts extracted from the corpus produced by Gungor. The results are satisfactory.

It will be very difficult to have any breakthroughs in authorship attribution studies if we are still confined to using traditional machine learning methods. I hope that neural network type methods, such as word2vec, can obtain breakthroughs. However,

authorship attribution is an unconventional branch of NLP, all papers and all software related to word2vec that I have encountered so far do not touch on authorship attribution. Therefore, I need to gain in-depth knowledge in word2vec if I want to adapt it to obtain a breakthrough in authorship attribution. To obtain in-depth knowledge of word2vec, I will first develop a word2vec programme for authorship attribution studies from scratch, by using basic Python.

**References**

Eisen M. et al. (2018) ‘Stylometric Analysis of Early Modern Period English Plays'\’, *Digital Scholarship in the Humanities*, 33(3), pp. 500–28.

Freebury-Jones, D. (2019) ‘“When a man hath a familiar style”: An introduction to authorship studies in early modern drama and literature’, *ANQ - Quarterly Journal of Short Articles Notes and Reviews*,

doi: 10.1080/0895769X.2019.1677210.

Gungor A. (2018) *Benchmarking authorship attribution techniques using over a thousand books by fifty Victorian Era novelists*. Unpublished MSc Thesis, Purdue University, Available at: https://scholarworks.iupui.edu/handle/1805/15938 (Accessed: 12 July 2020).

Kesidis G. and Walrand J. (1993) ‘Relative Entropy between Markov Transition Rate Matrices’, *IEEE Transactions on Information Theory*, 39(3), 1056–57

Segarra S. (2014) *Word adjacency networks for authorship attribution: solving Shakespearean controversies*. MSc Thesis, University of Pennsylvania, Available at: https://cpb-us-e1.wpmucdn.com/blogs.rice.edu/dist/a/9284/files/2018/07/MSc\_Thesis\_Segarra-zwwuyf.pdf (Accessed: 12 July 2020).

Segarra S. et al. (2013) ‘Authorship attribution using function words adjacency networks’, *International conference on acoustics, speech, and signal processing (ICASSP)*. Vancouver Convention and Exhibition Centre, Vancouver, 26-31 May. IEEE, pp. 5563–567.

Segarra S. et al. (2015) ‘Authorship attribution through function word adjacency networks’, *IEEE Transactions on Signal Processing*, 63(20), pp. 5464–478.

Segarra S. et al. (2016) ‘Attributing the Authorship of the Henry Vi Plays by Word Adjacency’, *Shakespeare Quarterly*, 67(2), pp. 232-256.

Weng, L. (2017) ‘Learning Word Embedding’, *Lil’log*, 15 October. Available at: https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html (Accessed: 28 August 2020).

**Appendix 1**

The code for applying KNN, SVM and neural network techniques on character 4-gram classifier to perform authorship attribution studies:

#DS7004 Gungor's dataset/ character 4-gram

#Three popular female novelists all born in the 1850s: 17 Helen Mathers 1853-1920 (18010- 18669 in the kaggle csv file), 32 Lucas Malet 1852-1931 (33861-34563), 33 Marie Corelli 1855-1924 (34564-36305)

#200 lines each

#there is a â in the code. If this code is loaded to RStudio, the encoding of it should be changed to UTF-8!!!

#set working directory and load package tm

setwd(dirname(file.choose()))

getwd()

if (!require('tm')) install.packages('tm'); library('tm')

#input data and form three dataframes

if(!file.exists('Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv')){

download.file('http://archive.ics.uci.edu/ml/machine-learning-databases/00454/dataset.zip', 'dataset.zip')

unzip('dataset.zip')

file.copy('./dataset/Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv', '.')

#if the working directory does not have the csv file, this if statement

#needs several minutes to run

}

dfVictorianEraAA <- read.table('Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv', header = TRUE, sep = (','), encoding = 'utf-8')

dfHelen\_Mathers18009\_18208 <- dfVictorianEraAA[18009:18208,]

dfLucas\_Malet33860\_34059 <- dfVictorianEraAA[33860:34059,]

dfMarie\_Corelli34563\_34762 <- dfVictorianEraAA[34563:34762,]

# Function for forming character 4-Grams

if (!require('stylo')) install.packages('stylo'); library('stylo')

library(stylo)

charNGramDf <- function(columnCell) {

my.text = gsub('\\s+', "\_", columnCell, perl = T)

my.vector.of.chars = txt.to.features(my.text, features = "c")

x = make.ngrams(my.vector.of.chars, ngram.size = 4)

xx = lapply(x,function(x) gsub('(?<=[\\S]) (?=[\\S])', '',x, perl = T))

return(paste(xx, collapse = ' '))

}

# Converting to 4-grams texts

dfHelen\_Mathers18009\_18208 <- as.data.frame(cbind(lapply(dfHelen\_Mathers18009\_18208[,1], charNGramDf), dfHelen\_Mathers18009\_18208[,2]))

colnames(dfHelen\_Mathers18009\_18208) <- c('text', 'author')

dfLucas\_Malet33860\_34059 <- as.data.frame(cbind(lapply(dfLucas\_Malet33860\_34059[,1], charNGramDf), dfLucas\_Malet33860\_34059[,2]))

colnames(dfLucas\_Malet33860\_34059) <- c('text', 'author')

dfMarie\_Corelli34563\_34762 <- as.data.frame(cbind(lapply(dfMarie\_Corelli34563\_34762[,1], charNGramDf), dfMarie\_Corelli34563\_34762[,2]))

colnames(dfMarie\_Corelli34563\_34762) <- c('text', 'author')

#form corpa from dataframes.

#texts are already all in lower case and no punctuation

#package tm is required

dfHelen\_Mathers18009\_18208\_corpus <- VCorpus(VectorSource(dfHelen\_Mathers18009\_18208$text))

dfHelen\_Mathers18009\_18208\_corpus <- tm\_map(dfHelen\_Mathers18009\_18208\_corpus, stripWhitespace)

dfLucas\_Malet33860\_34059\_corpus <- VCorpus(VectorSource(dfLucas\_Malet33860\_34059$text))

dfLucas\_Malet33860\_34059\_corpus <- tm\_map(dfLucas\_Malet33860\_34059\_corpus, stripWhitespace)

dfMarie\_Corelli34563\_34762\_corpus <- VCorpus(VectorSource(dfMarie\_Corelli34563\_34762$text))

dfMarie\_Corelli34563\_34762\_corpus <- tm\_map(dfMarie\_Corelli34563\_34762\_corpus, stripWhitespace)

#form dtm. Each line a document (1000 words)

#change minimum word length to 1 from 3

dfHelen\_Mathers18009\_18208\_dtDf <- as.data.frame(as.matrix(DocumentTermMatrix(dfHelen\_Mathers18009\_18208\_corpus, control=list(wordLengths = c(1, Inf)))))

dfLucas\_Malet33860\_34059\_dtDf <- as.data.frame(as.matrix(DocumentTermMatrix(dfLucas\_Malet33860\_34059\_corpus, control=list(wordLengths = c(1, Inf)))))

dfMarie\_Corelli34563\_34762\_dtDf <- as.data.frame(as.matrix(DocumentTermMatrix(dfMarie\_Corelli34563\_34762\_corpus, control=list(wordLengths = c(1, Inf)))))

#retain only columns of words which can found both in HM, LM and MC's texts

common\_cols <- intersect(intersect(colnames(dfHelen\_Mathers18009\_18208\_dtDf), colnames(dfLucas\_Malet33860\_34059\_dtDf)), colnames(dfMarie\_Corelli34563\_34762\_dtDf))

HmLmMcDtDf <- rbind(dfHelen\_Mathers18009\_18208\_dtDf[common\_cols], dfLucas\_Malet33860\_34059\_dtDf[common\_cols], dfMarie\_Corelli34563\_34762\_dtDf[common\_cols])#15220 cols

#delete columns with their names contain â #14924

HmLmMcDtDf <- HmLmMcDtDf[, -grep(pattern = '.\*â.\*â\*.\*', colnames(HmLmMcDtDf))]

#texts quite untidy. number of â in HM 2077, LM 1743 and MC 6280

#further retain only columns of words each of which are at least appeared

#600 times

HmLmMcTtl600OrMore <- HmLmMcDtDf[, colSums(HmLmMcDtDf) >=600] #975

#aggreate and sum every four lines (reduced to 150 lines)

#add and delete column textNO

HmLmMcTtl600OrMore$textNo <- rep(1:150, each = 4)

dfHmLmMcWdFeqDf <- aggregate(. ~ textNo, HmLmMcTtl600OrMore, sum)

dfHmLmMcWdFeqDf$textNo <- NULL

#add labels HM, LM and MC and put the column to the front

dfHmLmMcWdFeqDf$HmOrLmOrMc <- c(rep('HM', 50), rep('LM', 50), rep('MC', 50))

dfHmLmMcWdFeqDfLabled = dfHmLmMcWdFeqDf[,c(976,1:975)] #975+1

#shuffling rows:

set.seed(12345)

rrowNos <- sample(nrow(dfHmLmMcWdFeqDfLabled))

dfHmLmMcWdFeqDfLabledRandm <- dfHmLmMcWdFeqDfLabled[rrowNos,]

#normalisation

data\_norm <- function(x) {(x- min(x))/ (max(x)- min(x))}

dfHmLmMcWdFeqDfLabledRandm\_norm <- as.data.frame(lapply(dfHmLmMcWdFeqDfLabledRandm[,-1], data\_norm))

summary(dfHmLmMcWdFeqDfLabledRandm\_norm[,1:4]) #see whether normalised

View(dfHmLmMcWdFeqDfLabledRandm\_norm)

#KNN!

if (!require('class')) install.packages('class'); library('class')

dfHmLmMcWdFeqDfLabledRandm\_norm\_train <- dfHmLmMcWdFeqDfLabledRandm\_norm[1:120,]

dfHmLmMcWdFeqDfLabledRandm\_norm\_test <- dfHmLmMcWdFeqDfLabledRandm\_norm[121:150,]

HmOrLmOrMc\_pred <- knn(dfHmLmMcWdFeqDfLabledRandm\_norm\_train, dfHmLmMcWdFeqDfLabledRandm\_norm\_test, dfHmLmMcWdFeqDfLabledRandm[1:120,1], k= 11)

table(pred = HmOrLmOrMc\_pred, true\_HelenMathers\_LucasMalet\_MarieCorelli\_KNN = dfHmLmMcWdFeqDfLabledRandm[121:150,1]) #mistake rate 1/30

#sqrt(120) = 10.954 . Therefore use k =11.

#k = 11 perform the best, only one error: 1 MC was misjudged as LM

#SVM! tune automatically

if (!require('e1071')) install.packages('e1071'); library('e1071')

HmOrLmOrMc\_svm\_model <- svm(dfHmLmMcWdFeqDfLabledRandm\_norm\_train, as.factor(dfHmLmMcWdFeqDfLabledRandm[1:120,1]), type = 'C')

pred <- predict(HmOrLmOrMc\_svm\_model, dfHmLmMcWdFeqDfLabledRandm\_norm\_test)

table(pred, true\_HelenMathers\_LucasMalet\_MarieCorelli\_SVM = dfHmLmMcWdFeqDfLabledRandm[121:150,1])

#all correct

#tune manually

dfHmLmMcWdFeqDfLabledRandm1To120AsFactors = as.factor(dfHmLmMcWdFeqDfLabledRandm[1:120,1])

set.seed(12345)

svm\_tune <- tune(svm, train.x = dfHmLmMcWdFeqDfLabledRandm\_norm\_train,

train.y = dfHmLmMcWdFeqDfLabledRandm1To120AsFactors,

kernel = 'linear',

type = 'C',

ranges = list(cost = c(.001,.01,.1,1,5,10,100)))

svm\_tune

svm\_tune$best.model

#besides best cost, also best number of support vectors, etc.

pred\_svm\_after\_tune <- predict(svm\_tune$best.model, dfHmLmMcWdFeqDfLabledRandm\_norm\_test)

table(pred = pred\_svm\_after\_tune, true\_HelenMathers\_LucasMalet\_MarieCorelli\_TunedSVM = dfHmLmMcWdFeqDfLabledRandm[121:150,1])

# Deep learning using package keras

# import keras

# note: use\_condaenv("r\_reticulate") is only work for my Asus PC + Windows 10

# see README.md for matters related to installation of Python

if (!require('keras')) install.packages('keras'); library('keras')

use\_condaenv("r\_reticulate")

# Convert to matrix

training <- as.matrix(dfHmLmMcWdFeqDfLabledRandm\_norm[1:120,])

dimnames(training) <- NULL

test <- as.matrix(dfHmLmMcWdFeqDfLabledRandm\_norm[121:150,])

dimnames(test) <- NULL

# Convert labels to numerics and one hot encoding form

trainLabels <- to\_categorical(as.numeric(as.factor(dfHmLmMcWdFeqDfLabledRandm[1:120,1])) - 1)

testtarget <- as.numeric(as.factor(dfHmLmMcWdFeqDfLabledRandm[121:150,1])) - 1

testLabels <- to\_categorical(testtarget)

# Create sequential model (975 input columns, 3 categories)

model <- keras\_model\_sequential()

model %>% #one hidden layer, units = 975 (975 input columns, 3 categories)

layer\_dense(units=975, activation = 'relu', input\_shape = c(975)) %>%

layer\_dense(units=325, activation = 'relu', input\_shape = c(325)) %>%

layer\_dense(units = 3, activation = 'softmax')

summary(model) #see bottom for tunning results

# Compile

model %>%

compile(loss = 'categorical\_crossentropy',

optimizer = 'adam',

metrics = 'accuracy')

# Fit model

history <- model %>%

fit(training,

trainLabels,

epoch = 200,

batch\_size = 64,

validation\_split = 0.2)

# Prediction & confusion matrix - test data and labels

pred <- model %>%

predict\_classes(test)

library(caret)

confusionMatrix(table <- table(Predicted = pred, Actual = testtarget), mode = "everything")

#prob, pred, testtarget:

prob <- model %>%

predict\_proba(test)

cbind(prob, pred, testtarget)

#epoch 200 batch\_size 32 validation\_split 0.2 325 3

#f1: 0.9524 1 0.9474

#changed to 975 3 the same

**Appendix 2**

Interesting uses of word2vec

1) Test social bias of a corpus:

Below are the 20 words within a word embedding model produced by using a hotel review corpus and word2vec, cosine distances of them are closest to ‘man’ and ‘woman’ respectively. Do the texts contained in the corpus contain sex-biased elements?

Closest to ‘man’:

model.most\_similar("man")

[('woman', 0.628918468952179),

 ('lady', 0.5967980623245239),

 ('lad', 0.5614994168281555),

 ('monk', 0.5355309247970581),

 ('soldier', 0.5319280624389648),

 ('millionaire', 0.531794548034668),

 ('chap', 0.5119810104370117),

 ('farmer', 0.5109984278678894),

 ('guy', 0.5098308324813843),

 ('men', 0.5085940361022949)]

Closest to ‘woman’:  
model.most\_similar("woman")

[('lady', 0.6906163692474365),

 ('girl', 0.6630470156669617),

 ('prostitute', 0.6561852693557739),

 ('man', 0.6289184093475342),

 ('widow', 0.6273212432861328),

 ('nun', 0.6217451691627502),

 ('housewife', 0.6163227558135986),

 ('waitress', 0.5760902166366577),

 ('heiress', 0.5679841041564941),

 ('maid', 0.5663273334503174)]

Below are results obtained from the famous 3.39GB GoogleNews-vectors-negative300 (Google300) pretrained word embedding model:

Closest to ‘man’:

model.most\_similar("man")

[('woman', 0.7664012312889099),

 ('boy', 0.6824870109558105),

 ('teenager', 0.6586930751800537),

 ('teenage\_girl', 0.6147903203964233),

 ('girl', 0.5921714305877686),

 ('suspected\_purse\_snatcher', 0.571636438369751),

 ('robber', 0.5585119128227234),

 ('Robbery\_suspect', 0.5584409236907959),

 ('teen\_ager', 0.5549196004867554),

 ('men', 0.5489763021469116)]

Closest to ‘woman’:

model.most\_similar("woman")

[('man', 0.7664012312889099),

 ('girl', 0.7494640946388245),

 ('teenage\_girl', 0.7336829900741577),

 ('teenager', 0.631708562374115),

 ('lady', 0.6288785934448242),

 ('teenaged\_girl', 0.6141784191131592),

 ('mother', 0.607630729675293),

 ('policewoman', 0.6069462299346924),

 ('boy', 0.5975908041000366),

 ('Woman', 0.5770983099937439)]

2) Vector equation:

The results shown below represents a vector equation obtained from the Google300 pretrained word embedding model:

Oscar\_Wilde – man + woman ~= Jane\_Austen

>>>model.most\_similar(positive=['Oscar\_Wilde', 'woman'], negative=['man'])

>>>[('Jane\_Austen', 0.6260595321655273),

('Noël\_Coward', 0.600019097328186),

('Madame\_Bovary', 0.5772191286087036),

('Charlotte\_Bronte', 0.5637412667274475),

('Somerset\_Maugham', 0.5583583116531372),

('Noel\_Coward', 0.5572813749313354),

('Bernard\_Shaw', 0.5557636022567749),

('DH\_Lawrence', 0.5536727905273438),

('Antonia\_Fraser', 0.5471805334091187),

('An\_Ideal\_Husband', 0.541670024394989)]

lead – led + saw ~= see

>>>model.most\_similar(positive=['lead', 'saw'], negative=['led'])

>>>[('see', 0.4809260070323944),

('looked', 0.45732152462005615),

('seeing', 0.4199181795120239),

('advantage', 0.40815281867980957),

('knew', 0.3928772211074829),

('thought', 0.38903290033340454),

('noticed', 0.38654080033302307),

('squandered\_glorious', 0.3810598850250244),

('midway\_through', 0.37417805194854736),

('chances', 0.36055076122283936)]

Obtained from the word embedding model produced by my programme contained in Appendix 3, and the data of 2500 x 1000 words each from Charles Dickens, George Eliot and Jane Austen respectively:

king – man + woman ~= queen

>>>model.most\_similar(positive=['king', 'woman'], negative=['man']

>>> [('queen', 0.3741656541824341),

('saxon', 0.31361255049705505),

('throne', 0.3131310045719147),

('conqueror', 0.31052806973457336),

('earl', 0.29378655552864075),

('girl', 0.28226879239082336),

('child', 0.2698107063770294),

('dying', 0.26731109619140625),

('reign', 0.26275989413261414),

('pillow', 0.2603800594806671)]

better – good + bad ~= worse

>>>model.most\_similar(positive=['better', 'bad'], negative=['good'])

>>>[('worse', 0.38985705375671387),

('wiser', 0.37705114483833313),

('sooner', 0.3492392301559448),

('happier', 0.3015991449356079),

('bigger', 0.2892671227455139),

('fairer', 0.2862151265144348),

('apprehended', 0.2851130962371826),

('easier', 0.27977824211120605),

('fewer', 0.27307164669036865),

('liad', 0.260669469833374)]

3) Pick up the non-matching word:

>>>model.doesnt\_match("man woman child kitchen".split())

>>> 'kitchen'

**Appendix 3**

#LearnUsingGungorVictKMeanUseOwnEach2500LineTrain20Test8n14n26.py

# coding: utf-8

#!!!The data file Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv can be obtained from https://archive.ics.uci.edu/ml/machine-learning-databases/00454/

##input data

import pandas as pd

#note: use your own path

path\_to\_datafile = '..//..//DS7004//u1720146\_DS7004\_courseworkCodeAndData//preparationWorks//fromDS7003\_Gungor2018VictorianAuthorAttribution\_NGram//Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv'

pathToGungorVict = path\_to\_datafile

gungorVictRow = pd.read\_csv(pathToGungorVict, encoding = 'ISO-8859-1')

##form training data (2500 lines x 3) and test data (20 x 3)

##each line about 1000 words

#Use three authors' data:

#author:8 Charles Dickens total lines: 6914/ 14 George Eliot 2696/ 26 Jane Austen 4441

#each first 2500 lines for training, last 20 lines for testing. Each line has 1000 words

for i in [14, 26, 8]:

allLines = gungorVictRow.loc[gungorVictRow['author'] == i]

lines2500 = allLines.iloc[0:2500]

linesLast20 = allLines.iloc[-20:]

try:

train = train.append(lines2500)

test = test.append(linesLast20)

except:

train = lines2500

test = linesLast20

train = train.sample(frac=1, random\_state=42).reset\_index(drop = True) #7500 lines suffled

test = test.sample(frac=1, random\_state=42).reset\_index(drop = True) #60 lines suffled

## Import various modules for forming a string cleaning function

from bs4 import BeautifulSoup

import re

from nltk.corpus import stopwords

def text\_to\_wordlist( text, remove\_stopwords=False ):

# Function to convert a document to a sequence of words,

# optionally removing stop words. Returns a list of words.

#

# 1. Remove HTML

text = BeautifulSoup(text).get\_text()

#

# 2. Remove non-letters

text = re.sub("[^a-zA-Z]"," ", text)

#

# 3. Convert words to lower case and split them

words = text.lower().split()

#

# 4. Optionally remove stop words (false by default)

if remove\_stopwords: #These three lines will not be used. Pleasesee the second parameter of this function

stops = set(stopwords.words("english"))

words = [w for w in words if not w in stops]

#

# 5. Return a list of words

return(words)

## Download the punkt tokenizer and form a sentence splitting function

import nltk.data

#nltk.download() #no need to use this line again after it has been used once

# Load the punkt tokenizer

tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')

# Define a function to split a text into parsed sentences

def text\_to\_sentences( text, tokenizer, remove\_stopwords=False ):

# Function to split a text into parsed sentences. Returns a

# list of sentences, where each sentence is a list of words

#

# 1. Use the NLTK tokenizer to split the paragraph into sentences

raw\_sentences = tokenizer.tokenize(text.strip())

#

# 2. Loop over each sentence

sentences = []

for raw\_sentence in raw\_sentences:

# If a sentence is empty, skip it

if len(raw\_sentence) > 0:

# Otherwise, call text\_to\_wordlist to get a list of words

sentences.append( text\_to\_wordlist( raw\_sentence, remove\_stopwords )) #defined as false in text\_to\_wordlist

#

# Return the list of sentences (each sentence is a list of words,

# so this returns a list of lists

return sentences

#function for parsing the training set

def parsing\_sentence\_set(text\_df):

sentences = [] # Initialize an empty list of sentences

print("Parsing sentences from training set")

for text in text\_df["text"]:

sentences += text\_to\_sentences(text, tokenizer)

return sentences

##use the functions to form a cleaned unlabelled training set

##for performming unsupervised learning

sentences = parsing\_sentence\_set(train)

## Import the built-in logging module and configure it so that Word2Vec

# creates nice output messages

import logging

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)

# Set values for the single neural network layer's various parameters

#num\_features = 300 # Word vector dimensionality

#min\_word\_count = 40 # Minimum word count

#num\_workers = 4 # Number of threads to run in parallel

#context = 10 # Context window size

#downsampling = 1e-3 # Downsample setting for frequent words

num\_features = 300 # Word vector dimensionality

min\_word\_count = 5 # Minimum word count

num\_workers = 4 # Number of threads to run in parallel

context = 6 # Context window size

downsampling = 1e-3 # Downsample setting for frequent words

epochs= 20 #number of epochs

## Initialize and train the model (this will take some time)

# need to install gensim's word2vec

from gensim.models import word2vec

def form\_model\_from\_sentences(sentences):

print("Training model...")

model = word2vec.Word2Vec(sentences, workers=num\_workers, size=num\_features, min\_count = min\_word\_count, window = context, sample = downsampling, iter = epochs)

# If you don't plan to train the model any further, calling

# init\_sims will make the model much more memory-efficient.

model.init\_sims(replace=True)

return model

##form the word2vec model with the training set which will be

##used in the following two methods:

##vector averaging and vector clustering of stop words

model = form\_model\_from\_sentences(sentences)

##check the model

# king - man + woman ~= queen

print(model.most\_similar(positive=['king', 'woman'], negative=['man']))

##check the model

# better - good + bad ~= worse

model.most\_similar(positive=['better', 'bad'], negative=['good'])

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

##first method: vector averaging of stop words:

import gensim

all\_stopwords = set(gensim.parsing.preprocessing.STOPWORDS)

#be careful: nword and counter must be integers --Chiu

import numpy as np # Make sure that numpy is imported

def makeFeatureVec(words, model, num\_features):

# Function to average all of the word vectors in a given

# paragraph which are stop words

#

# Pre-initialize an empty numpy array (for speed)

featureVec = np.zeros((num\_features,),dtype="float32")

#

nwords = 0

#

# Index2word is a list that contains the names of the words in

# the model's vocabulary. Convert it to a set, for speed

index2word\_set = set(model.wv.index2word)

index2word\_set2 = all\_stopwords

#

# Loop over each word in the text and, if it is in the model's

# vocaublary and is a stop word add its feature vector to the total

for word in words:

if word in index2word\_set: #and word in index2word\_set2:

if word in index2word\_set2:

nwords = nwords + 1

featureVec = np.add(featureVec, model[word])

#

# Divide the result by the number of words to get the average

if nwords == 0:

nwords = 1 #avoid devided by zero (i.e. no stop word)

featureVec = np.divide(featureVec,nwords)

return featureVec

def getAvgFeatureVecs(texts, model, num\_features):

# Given a set of texts (each one a list of words), calculate

# the average feature vector for each one and return a 2D numpy array

#

# Initialize a counter

counter = 0

#

# Preallocate a 2D numpy array, for speed

textFeatureVecs = np.zeros((len(texts),num\_features),dtype="float32")

#

# Loop through the texts

for text in texts:

#

# Print a status message every 100th text

if counter%100 == 0:

haha = counter; hihi = len(texts)

print(f"Text {haha} of {hihi}") #% (counter, len(texts))

#

# Call the function (defined above) that makes average feature vectors

#textFeatureVecs[counter] = makeFeatureVec(text, model, num\_features)

textFeatureVecs[counter] = makeFeatureVec(text, model, num\_features)

# Increment the counter

counter = counter + 1

return textFeatureVecs

# Calculate average feature vectors for training and testing sets,

# using the functions we defined above.

clean\_train\_texts = []

for text in train["text"]:

#clean\_train\_reviews.append( review\_to\_wordlist( review, \

#remove\_stopwords=True )) #do not remove stop words

clean\_train\_texts.append( text\_to\_wordlist( text ))

trainDataVecs = getAvgFeatureVecs( clean\_train\_texts, model, num\_features )

print("Creating average feature vecs for test texts")

clean\_test\_texts = []

for text in test["text"]:

#clean\_test\_texts.append( text\_to\_wordlist( review, remove\_stopwords=True ))

clean\_test\_texts.append( text\_to\_wordlist( text ))

testDataVecs = getAvgFeatureVecs( clean\_test\_texts, model, num\_features )

# Fit a random forest to the training data, using 100 trees

from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier( n\_estimators = 100 )

print("Fitting a random forest to labeled training data...")

forest = forest.fit( trainDataVecs, train["author"] )

# Test & extract results

result = forest.predict( testDataVecs )

# Write the test results

output = pd.DataFrame( data={"true\_author":test["author"], "pred\_author":result} )

output.to\_csv( "Word2Vec\_AverageVectors.csv", index=False, quoting=3 )

confusion\_matrix = pd.crosstab(output['true\_author'], output['pred\_author'], rownames=['Actural'], colnames=['Predicted'])

print('Confusion matrix:\n', confusion\_matrix)

from sklearn import metrics

print('Accuracy: ', metrics.accuracy\_score(output['true\_author'], output['pred\_author']))

print('f1 score: ', metrics.f1\_score(output['true\_author'], output['pred\_author'], average = 'weighted'))

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

##second method: vector clustering of stop words (use KMeans):

from sklearn.cluster import KMeans

import time

start = time.time() # Start time (several to tens of minutes)

# Set "k" (num\_clusters) to be 1/5th of the vocabulary size, or an

# average of 5 words per cluster

word\_vectors = model.wv.syn0

num\_clusters = word\_vectors.shape[0] / 5

# Initalize a k-means object and use it to extract centroids

kmeans\_clustering = KMeans( n\_clusters = int(num\_clusters) )

idx = kmeans\_clustering.fit\_predict( word\_vectors )

# Get the end time and print how long the process took

end = time.time()

elapsed = end - start

print("Time taken for K Means clustering: ", elapsed, "seconds.")

# Create a Word / Index dictionary, mapping each vocabulary word to

#a cluster number

word\_centroid\_map = dict(zip( model.wv.index2word, idx ))

# For the first 10 clusters

for cluster in range(0,10):

#

# Print the cluster number

#print "\nCluster %d" #% cluster

print(f"\nCluster {cluster}")

#

# Find all of the words for that cluster number, and print them out

a\_view = word\_centroid\_map.items()

tuples = list(a\_view)

words = []

for i in range(0,len(word\_centroid\_map.values())):

if( tuples[i][1] == cluster ):

words.append(tuples[i][0])

print(words)

def create\_bag\_of\_centroids( wordlist, word\_centroid\_map ):

#

# The number of clusters is equal to the highest cluster index

# in the word / centroid map

num\_centroids = max( word\_centroid\_map.values() ) + 1

#

# Pre-allocate the bag of centroids vector (for speed)

bag\_of\_centroids = np.zeros( num\_centroids, dtype="float32" )

#

# Loop over the words in the review. If the word is in the vocabulary,

# find which cluster it belongs to, and increment that cluster count

# by one

for word in wordlist:

if word in word\_centroid\_map and word in all\_stopwords:

index = word\_centroid\_map[word]

bag\_of\_centroids[index] += 1

#

# Return the "bag of centroids"

return bag\_of\_centroids

# Pre-allocate an array for the training set bags of centroids (for speed)

train\_centroids = np.zeros( (train["text"].size, int(num\_clusters)), dtype="float32" )

# Transform the training set reviews into bags of centroids

counter = 0

for text in clean\_train\_texts:

train\_centroids[counter] = create\_bag\_of\_centroids( text, word\_centroid\_map )

counter += 1

# Repeat for test reviews

test\_centroids = np.zeros((test["text"].size, int(num\_clusters)), dtype="float32" )

counter = 0

for text in clean\_test\_texts:

test\_centroids[counter] = create\_bag\_of\_centroids( text, word\_centroid\_map )

counter += 1

# This cell take some minutes

# Fit a random forest and extract predictions

forest = RandomForestClassifier(n\_estimators = 100)

# Fitting the forest may take a few minutes

print("Fitting a random forest to labeled training data...")

forest = forest.fit(train\_centroids,train["author"])

result = forest.predict(test\_centroids)

# Write the test results

output = pd.DataFrame(data={"true\_author":test["author"], "pred\_author":result})

output.to\_csv( "BagOfCentroidsAuthor.csv", index=False, quoting=3 )

confusion\_matrix = pd.crosstab(output['true\_author'], output['pred\_author'], rownames=['Actural'], colnames=['Predicted'])

print('Confusion matrix:\n', confusion\_matrix)

from sklearn import metrics

print('Accuracy: ', metrics.accuracy\_score(output['true\_author'], output['pred\_author']))

print('f1 score: ', metrics.f1\_score(output['true\_author'], output['pred\_author'], average = 'weighted'))

**Appendix 4**

#LearnUsingGungorVictUseGoogleFixedAt485lines48n13n22n25n36.py

# coding: utf-8

#need to download GoogleNews-vectors-negative300.bin first

from gensim.models import Word2Vec, KeyedVectors

#use your path!!!

pathToGoogleNews300 = '..//fromBlogOfShaneLynnWordEmbeddingsWithSpacyAndGensim\_GoogleNews300//data//GoogleNews-vectors-negative300.bin'

model = KeyedVectors.load\_word2vec\_format(pathToGoogleNews300, binary=True)

import pandas as pd

#need to download Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv from http://archive.ics.uci.edu/ml/datasets/Victorian+Era+Authorship+Attribution

#use your path!!!

pathToGungorVict = '..//..//DS7004//u1720146\_DS7004\_courseworkCodeAndData//preparationWorks//fromDS7003\_Gungor2018VictorianAuthorAttribution\_NGram//Gungor\_2018\_VictorianAuthorAttribution\_data-train.csv'

gungorVictRow = pd.read\_csv(pathToGungorVict, encoding = 'ISO-8859-1')

#48: Washington Irving/ 13: Frances Hodgson Burnett/ 22: Jacob Abbott/ 25: James Payn/ 36: Oliver Optic

for i in [13, 22, 25, 36, 48]:

allLines = gungorVictRow.loc[gungorVictRow['author'] == i]

lines350 = allLines.iloc[0:350]

linesLast20 = allLines.iloc[-20:]

try:

train = train.append(lines350)

test = test.append(linesLast20)

except:

train = lines350

test = linesLast20

train = train.sample(frac=1, random\_state=42).reset\_index(drop = True) #1750 lines suffled

test = test.sample(frac=1, random\_state=42).reset\_index(drop = True) #100 lines suffledtest = authorsLines.sample(frac=0.3, random\_state=42)

print(model.most\_similar(positive=['king', 'woman'], negative=['man']))

print(model.most\_similar(positive=['better', 'bad'], negative=['good']))

print(model.most\_similar(positive=['lead', 'saw'], negative=['led']))

print(model.doesnt\_match("man woman child kitchen".split()))

import gensim

all\_stopwords = set(gensim.parsing.preprocessing.STOPWORDS)

#be careful: nword and counter must be integers --Chiu

import numpy as np # Make sure that numpy is imported

def makeFeatureVec(words, model, num\_features):

# Function to average all of the word vectors in a given

# paragraph

#

# Pre-initialize an empty numpy array (for speed)

featureVec = np.zeros((num\_features,),dtype="float32")

#

nwords = 0

#

# Index2word is a list that contains the names of the words in

# the model's vocabulary. Convert it to a set, for speed

index2word\_set = set(model.wv.index2word)

index2word\_set2 = all\_stopwords

#

# Loop over each word in the review and, if it is in the model's

# vocaublary, add its feature vector to the total

for word in words:

if word in index2word\_set:

if word in index2word\_set2:

nwords = nwords + 1

featureVec = np.add(featureVec,model[word])

#

# Divide the result by the number of words to get the average

if nwords == 0:

nwords = 1

featureVec = np.divide(featureVec,nwords)

return featureVec

def getAvgFeatureVecs(reviews, model, num\_features):

# Given a set of reviews (each one a list of words), calculate

# the average feature vector for each one and return a 2D numpy array

#

# Initialize a counter

counter = 0

#

# Preallocate a 2D numpy array, for speed

reviewFeatureVecs = np.zeros((len(reviews),num\_features),dtype="float32")

#

# Loop through the reviews

for review in reviews:

#

# Print a status message every 1000th review

if counter%50 == 0:

haha = counter; hihi = len(reviews)

print(f"Review {haha} of {hihi}") #% (counter, len(reviews))

#

# Call the function (defined above) that makes average feature vectors

#reviewFeatureVecs[counter] = makeFeatureVec(review, model, num\_features)

reviewFeatureVecs[counter] = makeFeatureVec(review, model, num\_features)

#

# Increment the counter

counter = counter + 1

return reviewFeatureVecs

# Import various modules for string cleaning

from bs4 import BeautifulSoup

import re

from nltk.corpus import stopwords

def review\_to\_wordlist( review, remove\_stopwords=False ):

# Function to convert a document to a sequence of words,

# optionally removing stop words. Returns a list of words.

#

# 1. Remove HTML

review\_text = BeautifulSoup(review).get\_text()

#

# 2. Remove non-letters

review\_text = re.sub("[^a-zA-Z]"," ", review\_text)

#

# 3. Convert words to lower case and split them

words = review\_text.lower().split()

#

# 4. Optionally remove stop words (false by default)

if remove\_stopwords:

stops = set(stopwords.words("english"))

words = [w for w in words if not w in stops]

#

# 5. Return a list of words

return(words)

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Calculate average feature vectors for training and testing sets,

# using the functions we defined above. Notice that we now use stop word

# removal.

num\_features = 300

clean\_train\_reviews = []

for review in train["text"]:

#clean\_train\_reviews.append( review\_to\_wordlist( review, \

#remove\_stopwords=True ))

clean\_train\_reviews.append( review\_to\_wordlist( review ))

trainDataVecs = getAvgFeatureVecs( clean\_train\_reviews, model, num\_features )

print("Creating average feature vecs for test reviews")

clean\_test\_reviews = []

for review in test["text"]:

#clean\_test\_reviews.append( review\_to\_wordlist( review, remove\_stopwords=True ))

clean\_test\_reviews.append( review\_to\_wordlist( review ))

testDataVecs = getAvgFeatureVecs( clean\_test\_reviews, model, num\_features )

# Fit a random forest to the training data, using 100 trees

from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier( n\_estimators = 100 )

print("Fitting a random forest to labeled training data...")

forest = forest.fit( trainDataVecs, train["author"] )

# Test & extract results

result = forest.predict( testDataVecs )

# Write the test results

output = pd.DataFrame( data={"true\_author":test["author"], "pred\_author":result} )

output.to\_csv( "Word2Vec\_AverageVectors.csv", index=False, quoting=3 )

confusion\_matrix = pd.crosstab(output['true\_author'], output['pred\_author'], rownames=['Actural'], colnames=['Predicted'])

print(confusion\_matrix)

from sklearn import metrics

print('Accuracy: ', metrics.accuracy\_score(output['true\_author'], output['pred\_author']))

print('f1 score: ', metrics.f1\_score(output['true\_author'], output['pred\_author'], average = 'weighted'))