UNIVERSITY OF BIRMINGHAM

The University of Birmingham

School of Computer Science

Assignment 2 – Semantic Embedding
Deadline: 17:00, December 09, 2022

Author:
Jizheng Wan

Reviewers/Markers:
Pieter Joubert
Jacqui Chetty
Ahmad Ibrahim
Madasar Shah

An Assignment submitted for the UoB:

Object Oriented Programming
Software Workshop I

December 02, 2022

Contents

Rules

1. Throughout the assignment, we refer to **attributes** although these can be called data fields or instance variables.
2. For each class refers to its corresponding test to verify field and method naming conventions.
3. Unless stipulated, all attributes are private, and all methods are public.
4. Although there are many ways to construct an application, you are required to adhere to the rules as stipulated below (to achieve marks).
5. If variable names are not stipulated, you can use your own names for variables. This shows that you have written the application (we will check for plagiarism).
6. You are not allowed to change the maximum Heap Size in IntelliJ. If you run into a memory problem, double-check your code.
7. Only the imports that we request you to provide are allowed. NO OTHER imports can be included.
8. Code efficiency is something we measure in this assignment. Your whole program (the *main*() class) should be able to finish within 2 seconds, the *WordsNearest*() (section 7.2 page 12, and 7.3 page 13) and *LogicalAnalogies*() (section 7.4, page 13) methods should take less than 100 milliseconds to execute. So, pay attention to the logical design and do NOT include unnecessary iterations in your methods.
9. Do NOT change or modify files included in the "resources" folder.
10. Do NOT modify the skeleton code.
11. Do NOT share your code with others.
12. This assignment is not as difficult as you might think (some of you might think it is too easy). You do not need to know how to use the GLOVE algorithm/method to produce the data file, and you do not need to know the reason behind the Distributional Semantic Model (DSM). If you are interested in these topics, then Natural Language Processing (NLP) is the module you should take. In this assignment, it is all about vector operations and their implementation in Java. Make sure you read this document carefully and interpret the requirements correctly.

HINT:

Pay special attention to the examples mentioned in Section 1.3, and the discussions in Section 5 if you don't know how to complete Task 6.

You can use the TODO window in IntelliJ (View | Tool Windows | TODO) to quickly jump between tasks.

1 Introduction

In this assignment, you need to build a prototype Question Answering system by using the semantic information embedded in a Distributional Semantic Model (DSM) file called "*glove*.6*B*.50*d_Reduced*.*csv*" (included in the "resources" folder).

For example, if "*London*" is the capital of the "*UK*", then we would like the system to identify that "*Beijing*" is the capital of "*China*"; or if the colour of "*Apple*" is "*Red*", then the colour of "*Banana*" should be "*Yellow*". Moreover, by providing a specific word, we would like the system to be able to find a list of words that are semantically close to it. E.g. By saying "*Computer*", the system should list "*Software*", "*Technology*", "*Internet*", "*Computing*" and "*Devices*" as the top 5 closest words. Or "*Doctorate*", "*Bachelor*", "*Thesis*", "*Dissertation*" and "*Graduate*" should be the top 5 closest words to "*PhD*". As mentioned already, we will use semantic embedding to achieve this, and a brief introduction will be provided below to help you better understand this assignment.

1.1 Word Embedding and GloVe

Computational linguistics research holds that word meaning can be represented by its contextual information because similar contextual distributions tend to share between semantically similar words [5]. The idea of Distributional Semantic Models (DSM), which have also been referred to as word space or vector space models, is that the meaning of words can, to a certain extent, be inferred from their usage and therefore the semantics can be encapsulated in high-dimensional vectors based on the nearby co-occurrence of words [7].

One of the most significant benefits of representing words with high-dimensional vectors is that the number-based representation can then be used as the input for further numerical processing, e.g. input for a neural network (NN). Hence, to a certain extent, DSM is simply a vectorisation or encoding process. But unlike the one-hot encoding, which simply assigns a unique number to each word in the vocabulary and the maximum number equals to the size of the vocabulary, the rationale behind DSM is to keep the original contextual and semantic information during the transformation process (also called word embedding/representation).

Many remarkable methods or frameworks have been developed in this area. For example, Word2Vec [1, 2, 4] and Global Vectors for Word Representation (GloVe) [6]. In this assignment, we are going to use a DSM generated by the GloVe method. You are not required to understand how GloVe works, but you do need to understand the structure of the DSM file.

1.2 DSM file

The file is called "*glove*.6*B*.50*d_Reduced*.*csv*" and is located in the "resources" folder. It was trained based on Wikipedia 2014 [1] + Gigaword 5 [2], which contains 6 billion tokens. Originally, there were 400,000 words included in this model. For demonstration

[1] https://dumps.wikimedia.org/enwiki/20140102/

[2] https://catalog.ldc.upenn.edu/LDC2011T07

purposes, we have reduced its size to only include 38,534 unique words. Below is an example of how this file is structured:

```
abacus,0.9102,-0.22416,0.37178,0.81798,...,0.34126
abadan,-0.33432,-0.95664,-0.23116,0.21188,...,-0.23159
abalone,0.34318,-0.8135,-0.99188,0.6452,0.0057126,...,-0.15903
   .
   .
   .
zygote,0.78116,-0.49601,0.02579,0.69854,...,-0.40833
zymogen,-0.34302,-0.76724,0.13492,-0.0059688,...,0.37539
```

Each line starts with a unique word (so 38,534 lines in total), then followed by 50 floating numbers (separated by ","). These floating numbers are the vector representation of that word. In other words, each unique word is associated with a size/length 50 vector. Elements in this vector must be consistent with the order of the floating numbers in the CSV file. Using the word "abacus" as an example, the first element in its vector representation should be "0.9102", then the second element is "-0.22416", and so on and so forth.

1.3 Extracting semantic information from vector representations

Based on the DSM theory, there is a clustering/grouping trend for words with similar meanings. Using the below figure (Fig. 1) [3] as an example, originally, it has been used to explain how Word2Vec understands semantic relationships, like Paris and France are related the same way Beijing and China are (capital and country), and not in the same way Lisbon and Japan are. Consequently, those words that represent the concept of “Country” have been grouped together and separated from those that represent the “Capital” concept.

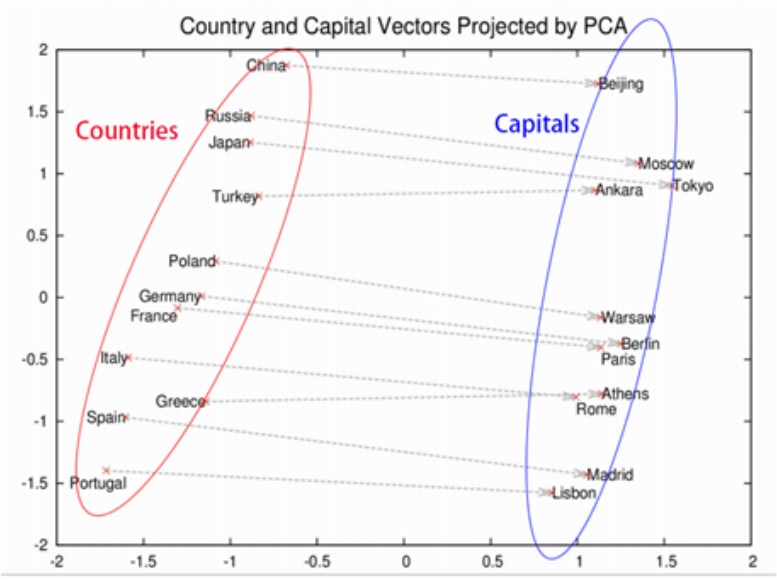

Figure 1: Clustering trend of vector representation

Moreover, the "closeness" of two words can be measured by *CosineSimilarity* (*CS*) (also called cosine distance). *CS* is a measure of the cosine of the angle between two non-zero vectors (θ in Fig. 2). It is calculated by using Equation 1 listed below:

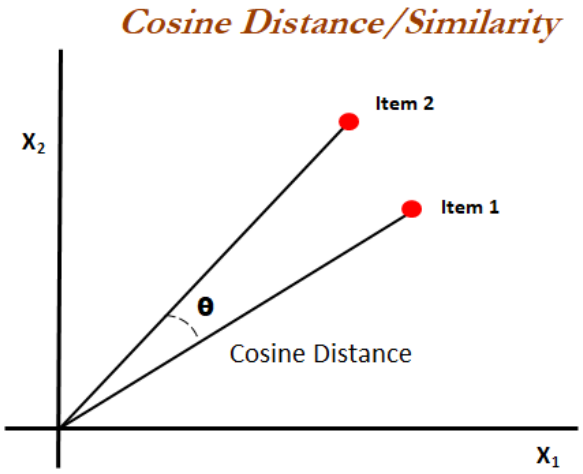

Figure 2: Cosine Similarity/Distance

$$
\begin{aligned}
CS(\overrightarrow{U}, \overrightarrow{V}) &= \frac{\overrightarrow{U} \cdot \overrightarrow{V}}{||\overrightarrow{U}||\,||\overrightarrow{V}||} \\
&= \frac{\sum_{i=1}^{n} u_i \times v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \times \sqrt{\sum_{i=1}^{n} v_i^2}}
\end{aligned} \tag{1}
$$

where $\overrightarrow{U}$ is the vector representation of item 1 (in our case, the vector representation of the first word) and $\overrightarrow{V}$ is the vector representation of item/word 2. u_i is the i-th element in $\overrightarrow{U}$, and v_i is the i-th element in $\overrightarrow{V}$.

So, back to the "*Computer*" examples mentioned at the beginning of the document. To identify the closest word to "*Computer*", we can calculate its *CS* value with all the other words included in the DSM file. Then the higher the *CS* value is, the closer a word is to "*Computer*". **This is, essentially, what you need to implement in Task 6.2 (Section 7.2, page 12)** .

Another interesting phenomenon is that vector operations can be used to interpret the relationship between words. For example, let $\overrightarrow{V}_{UK}$ be the vector representation of the word "*UK*", $\overrightarrow{V}_{London}$ be the vector representation of the word "*London*", $\overrightarrow{V}_{China}$ be the vector representation of the word "*China*" and $\overrightarrow{V}_{Beijing}$ be the vector representation of the word "*Beijing*", then $\overrightarrow{V}_{Beijing} - \overrightarrow{V}_{China} \approx \overrightarrow{V}_{London} - \overrightarrow{V}_{UK}$. In other words, $\overrightarrow{V}_{Beijing} \approx \overrightarrow{V}_{China} - \overrightarrow{V}_{UK} + \overrightarrow{V}_{London}$.

$\overrightarrow{V}_{China} - \overrightarrow{V}_{UK} + \overrightarrow{V}_{London}$ will produce a new vector, and it is highly unlikely that this new vector (denoted as $\overrightarrow{V}_{New}$) will have exactly the same elements as $\overrightarrow{V}_{Beijing}$.

However, we can overload the $WordsNearest(String\ \_word)$ method (produced in Task 6.2) to $WordsNearest(Vector\ \_vector)$ (**this is, essentially, what you need to implement in Task 6.3 (Section 7.3, page 13)**), and instead of using a String type argument we now pass $\overrightarrow{V}_{New}$ to the $WordsNearest(Vector\ \_vector)$ method to identify the closest word to this vector. Then this identified word would be "*Beijing*"! **This is, essentially, what you need to implement in Task 6.4 (Section 7.4, page 13)**.

2 Task 1 - Vector.java [26.5 Marks]

The Vector class consists of vector objects, and you need to complete the following methods to finish this class. Testing this class with the *VectorTest.java* file.

2.1 Task 1.1 - $Vector(double[]\ \_elements)$ [0.5 Marks]

This is the constructor of the Vector class.

- Complete this constructor by assigning the *_elements* to the *doubElements* attribute.

2.2 Task 1.2 - $getElementatIndex(int\ \_index)$ [2 Marks]

This is the method to get an element at a specific vector index.

- Complete this method to return an element in this vector based on the *_index*.
- If the index is out of bounds, return -1 instead.

2.3 Task 1.3 - $setElementatIndex(double\ \_value,\ int\ \_index)$ [2 Marks]

This is the method to set an element at a specific vector index.

- Set the value of an element at *_index* to *_value*.
- If the index is out of bounds, modify the value of the last element instead.

2.4 Task 1.4 - $getAllElements()$ [0.5 Marks]

- Return all elements in this vector.

2.5 Task 1.5 - $getVectorSize()$ [0.5 Marks]

- Return the size/length of this vector.

2.6 Task 1.6 - *reSize*(*int _size*) [6 Marks]

- If *_size* equals to the current length of the vector, or *_size* $<= 0$ return the existing vector without any modifications.
- Otherwise, returns a new vector with the specified size/length:
  - If *_size* is smaller than the current length, only keep the first X elements (X=*_size*). For example, resizing [1.0,2.0,3.0,4.0,5.0] to 3 should give you [1.0,2.0,3.0].
  - Otherwise, assign -1.0 to the new elements added to the vector but existing elements remain unchanged. For example, resizing [1.0,2.0,3.0] to 5 should give you [1.0,2.0,3.0,-1.0,-1.0];

2.7 Task 1.7 - *add*(*Vector _v*) [2 Marks]

- Adding another vector *_v* to the current vector.
- If the length of *_v* is bigger than the length of the current vector, call the *reSize*(*int _size*) method to increase the length of the current vector (both vectors should have the same length);
- Otherwise, increase the length of *_v* and make it equal to the length of the current vector.

The formula of vector addition:

$$\overrightarrow{U} + \overrightarrow{V} = [u_1 + v_1,\ u_2 + v_2,\ ...,\ u_i + v_i] \tag{2}$$

where u_i is the i-th element in the vector $\overrightarrow{U}$, v_i is the i-th element in the vector $\overrightarrow{V}$.

2.8 Task 1.8 - *subtraction*(*Vector _v*) [2 Marks]

- Subtracting another vector *_v* to the current vector.
- If the length of *_v* is bigger than the length of the current vector, call the *reSize*(*int _size*) method to increase the length of the current vector (both vectors should have the same length);
- Otherwise, increase the length of *_v* and make it equal to the length of the current vector.

The formula of vector subtraction:

$$\overrightarrow{U} - \overrightarrow{V} = [u_1 - v_1,\ u_2 - v_2,\ ...,\ u_i - v_i] \tag{3}$$

where u_i is the i-th element in the vector $\overrightarrow{U}$, v_i is the i-th element in the vector $\overrightarrow{V}$.

2.9 Task 1.9 - $dotProduct(Vector\ \_v)$ [2 Marks]

- Dot product the current vector with vector $\_v$.
- As with the previous method, call the $reSize(int\ \_size)$ method to make them having the same length.

The formula of dot product:

$$\overrightarrow{U} \cdot \overrightarrow{V} = \sum_{i=1}^{n} u_i \times v_i \tag{4}$$

where u_i is the i-th element in the vector $\overrightarrow{U}$, v_i is the i-th element in the vector $\overrightarrow{V}$.

2.10 Task 1.10 - $cosineSimilarity(Vector\ \_v)$ [6 Marks]

- Calculate the *CS* value between the current vector and vector $\_v$.
- As with the previous method, call the $reSize(int\ \_size)$ method to make them having the same length.

The formula of Cosine Similarity was introduced already - Equation 1, page 6.

2.11 Task 1.11 - $equals(Object\ \_obj)$ [3 Marks]

- Override the *.equals* method. Return "true" if all elements in these two vectors are the same. Otherwise, return "false".
- If they have a different length, return "false";

3 Task 2 - Glove.java [2.5 Marks]

The Glove class consists of GloVe objects, and you need to complete the following methods to finish this class. *strVocabulary* is the attribute of the word stored in this Glove object, and *vecVector* is its vector representation.

Testing this class with the *GloveTest.java* file.

3.1 Task 2.1 - $Glove(String\ \_vocabulary, Vector\ \_vector)$ [0.5 Marks]

This is the constructor of the Glove class.

- Complete this constructor by assigning the *_vocabulary* to the *strVocabulary* attribute, and *_vector* to *vecVector*.

3.2 Task 2.2 - Task 2.5 [0.5 Marks each]

- Complete the relevant get and set methods accordingly.

4 Task 3 - CosSimilarityPair.java [5 Marks]

The CosSimilarityPair class consists of Cosine Similarity pair objects. A Cosine Similarity pair is the Cosine Similarity value (*doubCS*) between either two words (*strWord*1 and *strWord*2) or between a vector (*vecVector*) and a word (*strWord*2). The former is used in the *WordsNearest*(*String _word*) method in the *SemanticMain.java* file (Task 6.2), and the latter is used in the *WordsNearest*(*Vector _vector*) method in the same file (Task 6.3).

You need to complete the following methods to finish this class and test this class with the *CosSimilarityPair.java* file.

4.1 Task 3.1 - Task 3.2 *CosSimilarityPair*() [0.5 Marks each]

- Complete these two constructors accordingly.

4.2 Task 3.3 - Task 3.10 [0.5 Marks each]

- Complete the relevant get and set methods accordingly.

5 Task 4 - HeapSort.java [10 Marks]

HeapSort is one of the most efficient comparison-based sorting algorithms in computer science[3]. As explained previously, you need to compare the Cosine Similarity value in various pairs to determine the semantic closeness. Hence, an efficient sorting method is essential to this assignment.

Followed by the "Computer" example discussed in Section 1.3, you will need to call the *WordsNearest*(*String _word*) method (and pass the word "computer" into it) to calculate the Cosine Similarity value between "computer" and the other words in the DSM file and initialise an associated *CosSimilarityPair* (which was completed in Task 3) object after each calculation. You can then create a List or ArrayList (e.g. *List* $<$ *CosSimilarityPair* $>$) to store these objects. The idea here is to pass this unsorted list into the HeapSort class to get it sorted (descending) based on the *CS* values. Then the first object in this list will be the pair with the highest *CS* value, and the *strWord*2 (remember that "Computer" needs to be the *strWord*1) in that pair will be the closest word to "Computer". If you need to find the top five words, simply retrieve the *strWord*2 from the first five objects in that sorted list.

[3] https://en.wikipedia.org/wiki/Heapsort

You do not need to know how HeapSort works, since the code is provided already. However, the provided code only accepts *int*[] as an input. Hence, your task is to overload these existing methods (in the HeapSort class), so that they can accept $List < CosSimilarityPair >$ as an input. Please refer to the *HeapSortTest.java* file if you want to know how it is supposed to work.

5.1 Task 4.1 - $doHeapSort(List < CosSimilarityPair > \_list)$ [1 Mark]

- Complete this method based on how it's been implemented in $doHeapSort(int[]\ \_arr)$.

5.2 Task 4.2 - $heapSort(List < CosSimilarityPair > \_list)$ [4 Mark]

- Complete this method based on how it's been implemented in $heapSort(int[]\ \_arr)$.

5.3 Task 4.3 - $heapify(List < CosSimilarityPair > \_tree, ...)$ [5 Mark]

- Complete this method based on how it's been implemented in $heapify(int[]\ \_tree, ...)$.

6 Task 5 - Toolkit.java [10 Marks]

The Toolkit class includes methods you need to use/complete to load data from the DSM file.

FILENAME_GLOVE is the name of the DSM file (the structure of this file can be found in Section 1.2, page 4), and *FILENAME_STOPWORDS* is the name of the file that contains a list of stop words (will explain the usage of it in Task 6.1).

HINT: Remember to use the *try...catch()...finally* blocks. Do NOT hardcode your file path.

6.1 Task 5.1 - $loadGLOVE()$ [5 Marks]

In this task, you are required to use a *BufferedReader* (*myReader*) to read data from the DSM file (*FILENAME_GLOVE*) line by line.

- Read the file line by line and analyse the result - adding the word to *listVocabulary* and its vector representation to *listVectors*.
- Use the $Toolkit.getFileFromResource(String\ \_fileName)$ method to get the correct file path.

6.2 Task 5.2 - $loadStopWords()$ [5 Marks]

Similar to Task 5.1, used a *BufferedReader* to read the *FILENAME_STOPWORDS* file, and add those stop words in *listStopWords* defined in this method.

- Use the $Toolkit.getFileFromResource(String\ \_fileName)$ method to get the correct file path.
- Return the *listStopWords* list at the end of this method.

7 Task 6 - SemanticMain.java [46 Marks]

Within the constructor, it will call the $Toolkit.loadStopWords()$ method and use the returned value to populate the *STOPWORDS* attribute. Then call the $Toolkit.loadStopWords()$ method to assign values to the *listVocabulary* and *listVectors* defined in the *Toolkit* class.

Please use the provided *main*() method to figure out how we would like this application to work.

7.1 Task 6.1 - $CreateGloveList()$ [4 Marks]

This method is used to populate the *listGlove* attribute based on *listVocabulary* and *listVectors*.

- In NLP, Stop Words are insignificant words with little semantic meaning. Hence, quite often they will be removed from the related NLP task. In our case, please skip the Glove initialisation if a specific word (within the *listVocabulary*) is a Stop Word. In other words, if a word exists in the *STOPWORDS* list, do not create a Glove object.
- After filtering out all the Stop Words, add all the initialised Glove objects to the *listGlove*.

7.2 Task 6.2 - $WordsNearest(String\ \_word)$ [15 Marks]

Based on what we have discussed so far, calculate the Cosine Similarity (by using the *cosineSimilarity*() method defined in the *Vector* class) between a specified word (*_word*) and the other words in the *listGlove*.

HINT: We have removed stop words when creating the *listGlove*. Hence, use *listGlove* to conduct the loop instead of *listVocabulary*.

- For a specific word, do NOT calculate the *CS* value with itself.
- If the specified word does not exist, use the word "error" instead.
- As explained in Section 5, use the *listCosineSimilarity* defined in this method to store the *CosSimilarityPair* object created after each calculation. Then use the *HeapSort* class to sort *listCosineSimilarity* and return the sorted list.

7.3 Task 6.3 - $WordsNearest(Vector\ \_vector)$ [15 Marks]

Overload the $WordsNearest(String\ \_word)$ method, and use a vector parameter ($\_vector$) instead of a String to receive a similar sorted list.

- If $\_vector$ is the vector of an existing word (use the *equals* method defined in the $Vector$ class) in $listGlove$, then do NOT calculate the CS value with itself (and generate a $Glove$ object).
- Otherwise, please refer to the examples (especially $\overrightarrow{V}_{New}$) discussed in Section 1.3 to figure out what to do.

7.4 Task 6.4 - $LogicalAnalogies(String\ \_secISRef, ...)$ [12 Marks]

This method is used to calculate logical analogies.

- For example, "uk is to london as china is to beijing". The word "uk" should be used as the $\_firISRef$ parameter in this method, "london" is the $\_firTORef$ parameter and "china" should be the $\_secISRef$. Again, please refer to the examples in Section 1.3 and the discussions in Section 5 to figure out how to complete this method to identify the word "beijing".
- The last parameter ($\_top$) in this method is used to control the top X ($X = \_top$) results you would like to return. Hence, the size of $listResult$ should be $\_top$;
- Excluding pairs that contain $\_firISRef$, $\_firTORef$ or $\_secISRef$ in the final list ($listResult$).

HINT: Remember to use the $WordsNearest(Vector\ \_vector)$ method.

References

[1] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, 1 2013. [Online]. Available: http://ronan.collobert.com/senna/

[2] T. Mikolov, Q. V. Le, and I. Sutskever, "Exploiting similarities among languages for machine translation," *ArXiv*, 9 2013. [Online]. Available: http://arxiv.org/abs/1309.4168

[3] T. Mikolov, I. Sutskever, and L. L. Quoc, "Learning the meaning behind words," *Google Open Source Blog*, 2016. [Online]. Available: https://opensource.googleblog.com/2013/08/learning-meaning-behind-words.html

[4] T. Mikolov, W. T. Yih, and G. Zweig, "Linguistic regularities in continuous space-word representations," *NAACL HLT 2013 - 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Main Conference*, pp. 746–751, 2013.

[5] G. A. Miller and W. G. Charles, "Contextual correlates of semantic similarity," *Language and Cognitive Processes*, vol. 6, pp. 1–28, 1 1991. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01690969108406936

[6] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[7] S. Pulman, "Distributional semantic models," *Quantum Physics and Linguistics: A Compositional, Diagrammatic Discourse*, pp. 333–358, 2 2013. [Online]. Available: https://oxford.universitypressscholarship.com/view/10.1093/acprof:oso/9780199646296.001.0001/acprof-9780199646296-chapter-12