CS565 Assignment 2

MegaMind

12 March 2017

1 Neural Probabilistic Language Model

This section was done by Mrinal Tak.

For this question we used the Neural Probabilistic Language Model which was defined by Yoshua Bengio [1]. We briefly describe the model in this section. We have a training set, which is a sequence of words $w_1, w_2.....w_T$ where $w_t \in V$.

We want to learn $f(w_t, ..., w_{t-n+1}) = \hat{P}(w_t|w_1^{t-1})$.

The word embeddings are initialized randomly, and the vectors corresponding to the input window are concatenated and a *tanh* non-linear activation is performed before being fed into the hidden layer. The hidden layer takes this window vectors as input and output a score for each word in the vocabulary. A softmax layer then converts these scores into a corresponding probability distribution.

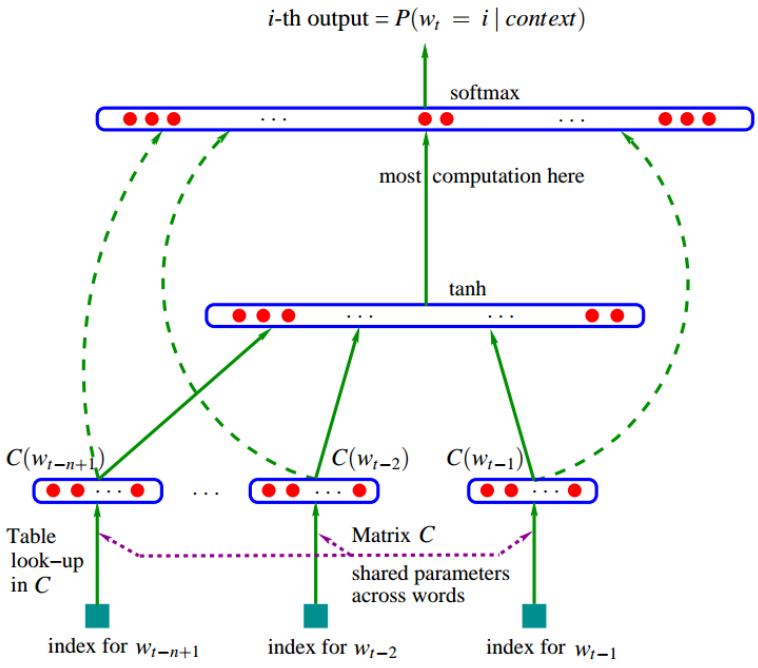


Figure 1: The Model

[1] Bengio, Yoshua, et al. "A neural probabilistic language model." Journal of machine learning research. Feb (2003): 1137-1155.

The input to the model will be a sequence of 5 words, $w_1, w_2, \ldots, w_5$. We pass it through the first hidden layer to obtain x=$(C(w_1), C(w_2).....C(w_5))$ which is a concatenation of the embeddings of the input words.

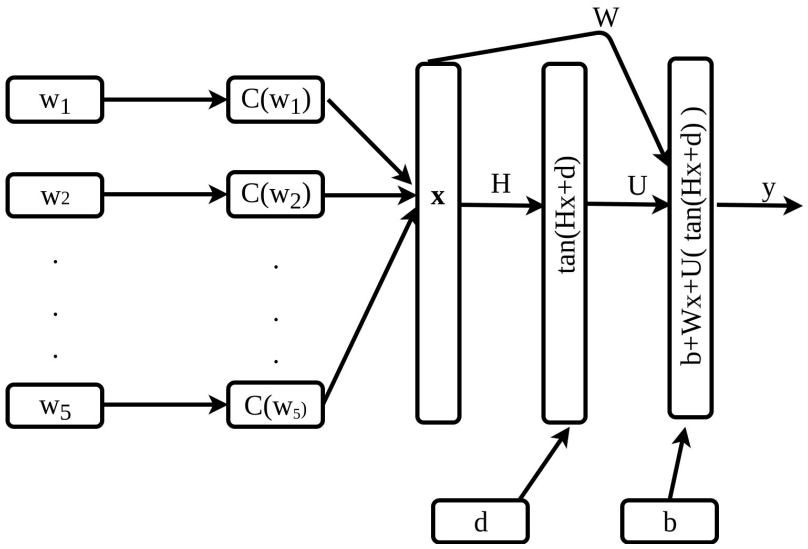


Figure 2: Neural architecture

1.1 Implementation details

- We have chosen only the 10000 most frequent words from the corpus and the rest of the words are replaced by UNK.
- We have trained our model for 5 epochs and in batches of size 5000 to obtain the embedding Matrix which is saved in `embeddings/emb_nplm_5epochs.txt`.
- A cross-entropy loss is used as the objective function, and gradient descent approach is used to minimize the loss.
- A regularization parameter of 0.01 is used with the weight parameters in each layer.

2 Latent Semantic Analysis

This section is done by Sumeet Ranka.

Latent Semantic Analysis (LSA) is generally used in finding relationships between words and a set of topics. The assumption is that words that have similar meanings will appear in similar texts or will be used in texts speaking about the same topic.

Here in this assignment we are using a word-word co-occurrence matrix to find word embeddings and analyse the similarity between the words. The idea is that for a given window-width we find the number of co-occurrences between a pair of words and represent this data in a matrix form. This obtained matrix is then put to Singular Value Decomposition (SVD) to obtain word embeddings.

Applying SVD to X:

$$\underset{|V|}{\begin{bmatrix} \\ X \\ \\ \end{bmatrix}} = \underset{|V|}{\begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix}} \underset{|V|}{\begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}} \underset{|V|}{\begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix}}$$

Reducing dimensionality by selecting first k singular vectors:

$$\underset{|V|}{\begin{bmatrix} \\ \hat{X} \\ \\ \end{bmatrix}} = \underset{|V|}{\begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix}} \underset{k}{\begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}} \underset{|V|}{\begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix}}$$

Figure 3: Singular Value Decomposition

Refer to Figure 3 for a pictorial representation on how it is performed.
Values of hyper-parameters while performing this experiment:

1. Window Width = 5
2. Shape of a Word Vector (k) = (50,1)
3. Size of Vocabulary = 10000 (Top 9999 most frequent words are chosen and rest are replaced with the token 'UNK')

The experiment is performed using two methods:

- Using a library
- Implementing from the scratch

2.1 Using a library

A combination of libraries were used to compute the truncated SVD matrix. A python library named `textacy` was used both in calculating the co-occurrence matrix as well the truncated matrix. However for the calculation of the adjacency matrix a python library `networkx` was used and for truncated matrix `textacy` internally uses the `TruncatedSVD` method of `scikit-learn` python library.

Calculation of Co-Occurrence Matrix

There are some in-built functions in library that can be used to create a semantic network graph for words. The function `terms_to_semantic_network` takes a list of terms and a window width to create a graph where each node represents for a word and an edge between them represents the association between those two words. In the case of this experiment the edge represented the co-occurrence of the two words in a given window and the edge weight is representing the frequency of that co-occurrence.

The graph obtained is converted to an adjacency matrix using a function `adjacency_matrix` of `networkx` library. This matrix obtained is our co-occurrence matrix where an element of cell, say `i,j` represents the number of times word_i appears in the context of word_j. Hence, we can also

observe that this is going to be a symmetric matrix. As our vocabulary size is 10000 so the matrix obtained will be of shape (10000, 10000).

Truncated SVD from Co-Occurrence Matrix

The library `textacy` contains functions that internally calls the implemented functions of various topic models in `scikit-learn` library. By giving 50 as an argument for number of topics we call the LSA function that returns the truncated matrix. It takes two other arguments, type of algorithm to be used and number of iterations for which you want to run the algorithm. The library has support for two algorithms: a fast randomised SVD solver[2] and a naive algorithm that would solve eigenvalues and eigenvectors for $X \times X^{\mathrm{T}}$ and $X^{\mathrm{T}} \times X$ and get the matrix.

We have used the first one because for larger matrices it is more efficient and the iterations was specified to 10 as opposed to default value of 5. This SVD algorithm is applied on a matrix of size (10000, 10000) and it returns a matrix of size (10000, 50). The function has been developed to parallelise the computations involved and use all the CPUs available in the host computer. As a result we can notice the the usage of CPU resource jumping to 100%.

2.2 Implementing from the scratch

The implementations in the library have been code well enough to use the available resources efficiently. The implementation by us could not match those standards but some efforts have been to put to match them.

Calculation of Co-Occurrence Matrix

Firstly, the ordered set of terms is converted to a list of context windows where every element of the list itself is a list of size equal to that of window width and contains terms in that window. Then the list of contexts are iterated over to count the number of number of co-occurrences for every pair of words and the count is updated in the co-occurrence matrix.

The code implementation for calculation of co-occurrence matrix can be found in the definition of the function named `terms_to_adj_matrix` in the file `coOccure.py`. To avoid the calculation of this matrix again and again this matrix has been pickled and stored in a file named `codedCoMatrix.p` inside the data folder. This can be directly loaded to proceed with the subsequent calculations or experiments.

As it is a deterministic algorithm and no approximation has been done the matrices obtained in both the cases will be same.

Truncated SVD from Co-Occurrence Matrix

For the calculation of truncated matrix an algorithm known as Power Method has been used. It is a classical eigenvalue algorithm[3].

There are two main factors that affects the time complexity of this algorithm:

- Size of the matrix: It involves a multiplication between a matrix and its transpose.
- Difference in eigenvalues: Determines the rate of convergence.

[2]Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 (arXiv:909) http://arxiv.org/pdf/0909.4061

[3]http://emis.ams.org/journals/ASUO/mathematics_/anale2015vol2/Bentbib_A.H.__Kanber_A..pdf

The implementation of the algorithm can be found in the file named `self_svd.py`. It contains two functions: `svd_1d` and `svd`.

`svd_1d` function contains the main bottleneck of the implementation where the matrix multiplication takes place. This function returns the singular vectors for the matrix. This function gets called 50 times in our implementation because we want to reduce the embedding to dimension of (50, 1).

`svd` function is the caller of the function `svd_1d`. Its main objective is to combine the singular vectors obtained and return the matrices U, V and Σ.

As we cannot control the eigenvalues as it depends on the matrix but we can control the efficiency with which the matrix multiplication can be done. As it is easy to solve this concurrently, hence this computation was performed over all the available processors.

2.3 Comparison between the two word embeddings

The comparison between the two different implementations of SVD are explained by taking 100 words as reference. These words chosen are the top 20 terms from each of the 50 topics. The list of the words chosen can be found in the file `words.tsne`.

The word embeddings for all 10000 words can be found in these files: `lib_svd.embed` and `self_svd.embed`.

The topics vs their top 20 words can be found in the file `svd.clusters`.

Some of the observations have been depicted in Figure 4. The figures 4(a), 4(c), 4(e) and 4(f) depicts projected embedding for method using library and figures 4(b), 4(d) and 4(g) indicates for implementation from the scratch. It is observable that both the methods are able to group together words that are similar in meaning. However the library method seems to do the job much better than our method. It gives more accurate result. The vectors are similar in both the cases but the cosine similarity between similar words seems to have more value in case of method using the library. For eg:

Cosine Similarity for words (**east**, **west**) in method 1: 0.895

Cosine Similarity for words (**east**, **west**) in method 2: 0.623

It is also interesting to observe from the images 4(h) and 4(i) that words that are collocated with the same words get grouped together. This seems obvious because we have considered co-occurrence frequency. Hence, say **nations** and **kingdom**, can be interpreted as similar in this method of finding similarity as they occur with the word **new** very often.

These observations are comparison between the two methods of implementation. More further observations have been mentioned and discussed in the section 4.1.

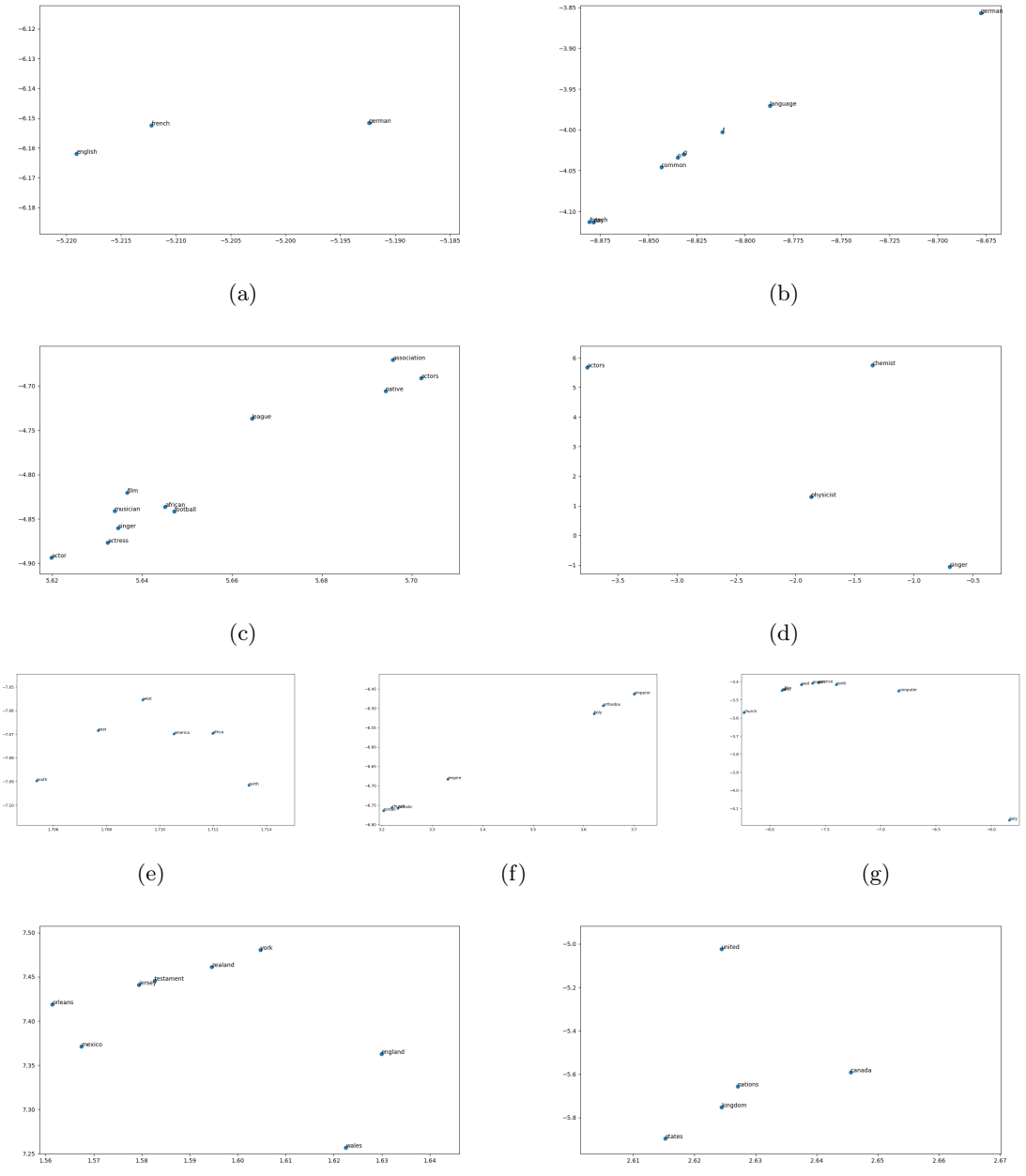
3 Word2Vec

This part was done by Desh Raj, with minor inputs from Mrinal Tak.

In the continuous bag-of-words architecture [4], the model predicts the current word from a window of surrounding context words. The order of context words does not influence prediction.

The word vectors are initialized randomly. In the input layer, we take the one-hot encoding of the words in the context window and get the corresponding vectors through an embedding lookup. The mean of these vectors is then fed into the hidden layer.

[4]Rong, Xin. "word2vec parameter learning explained." arXiv preprint arXiv:1411.2738 (2014).



(a) (b)

(c) (d)

(e) (f) (g)

(h) (i)

Figure 4: Projections of word embedding calculated using both the methods

Negative sampling is used for learning. For this purpose, the counts of all the vocabulary words is fed into the model. The model then generates the unigram probability distribution and this distribution is used to sample the negative words randomly.

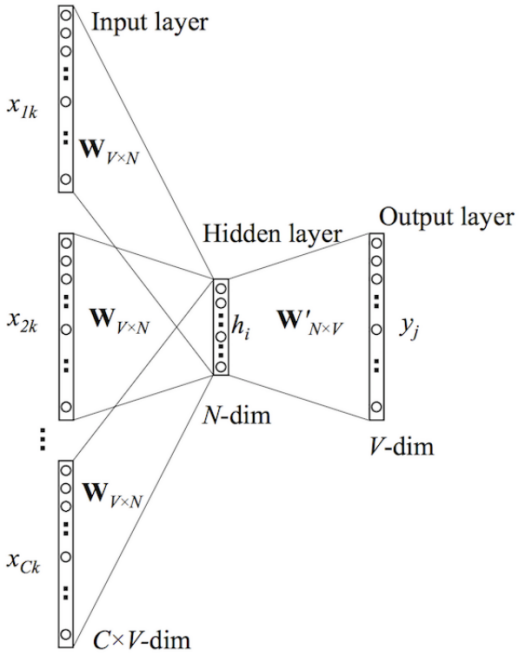


Figure 5: architecture of the CBOW Model

3.1 Implementation details

- In the actual implementation, the **nce_loss** inbuilt function in Tensorflow was used. It has the negative sampling integrated within it.
- A window size of 5 and a negative sample size of 10 was used.
- Gradient descent method was used for minimization of the loss function.
- The embeddings were trained for 50 epochs with a batch size of 5000. The obtained embeddings are saved in `embeddings/emb_w2v_50epochs.txt`.

4 Evaluations of Word Embedding

4.1 Using t-SNE

This part was done jointly by Sumeet Ranka, Venkat Arun, and Mrinal Tak.

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets. We applied t-SNE to analyze the embeddings for:

- NPLM model
- LSA model
- CBOW model

The t-SNE projections obtained are shown in the following figures.

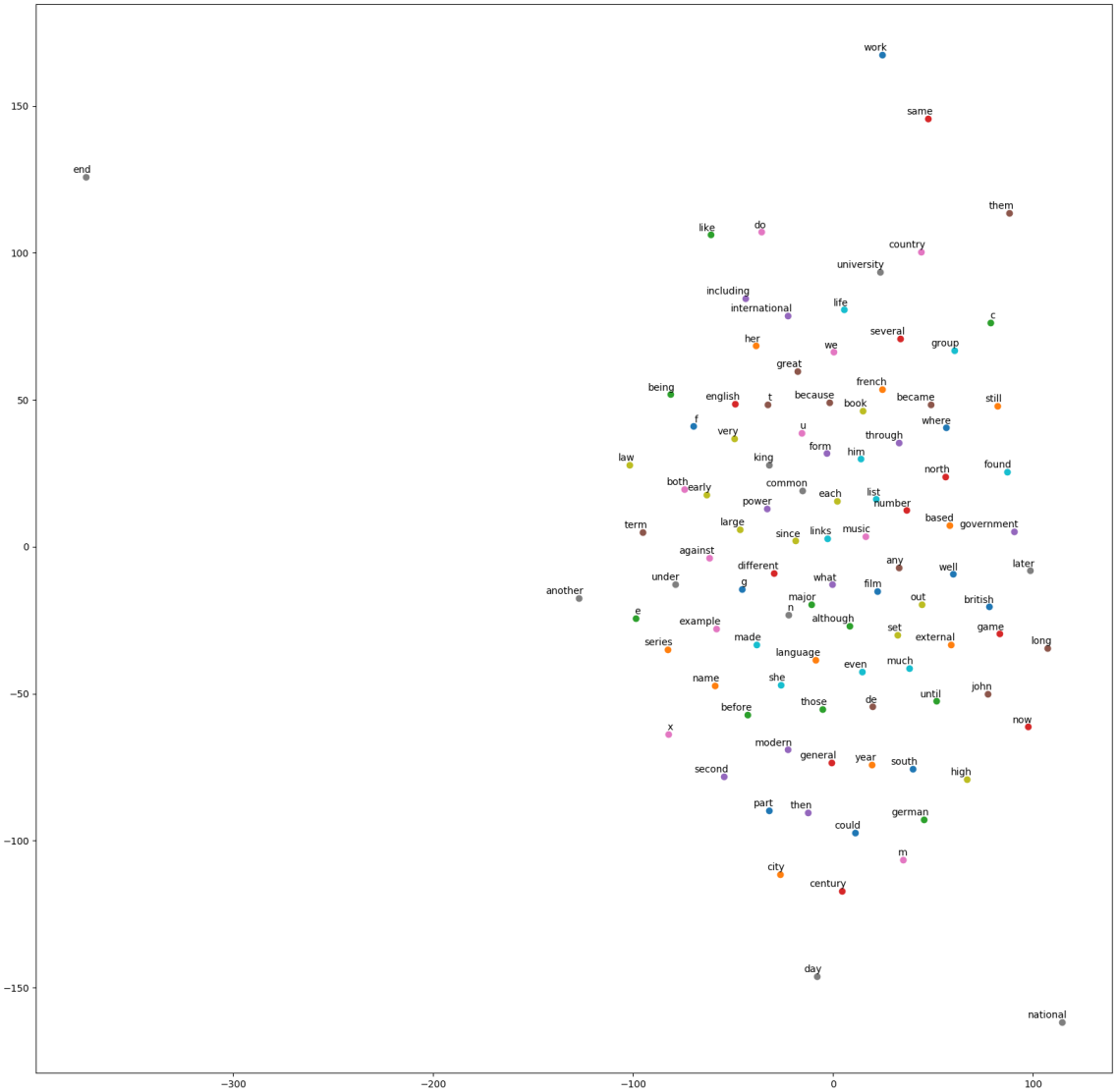


Figure 6: t-SNE projections of NPLM Model

Figure 7: In the NPLM model, we observe that certain words like king,power,large,law appear together and words like english,french,book appear together. We also observe that government is close to british, king ,french and power.

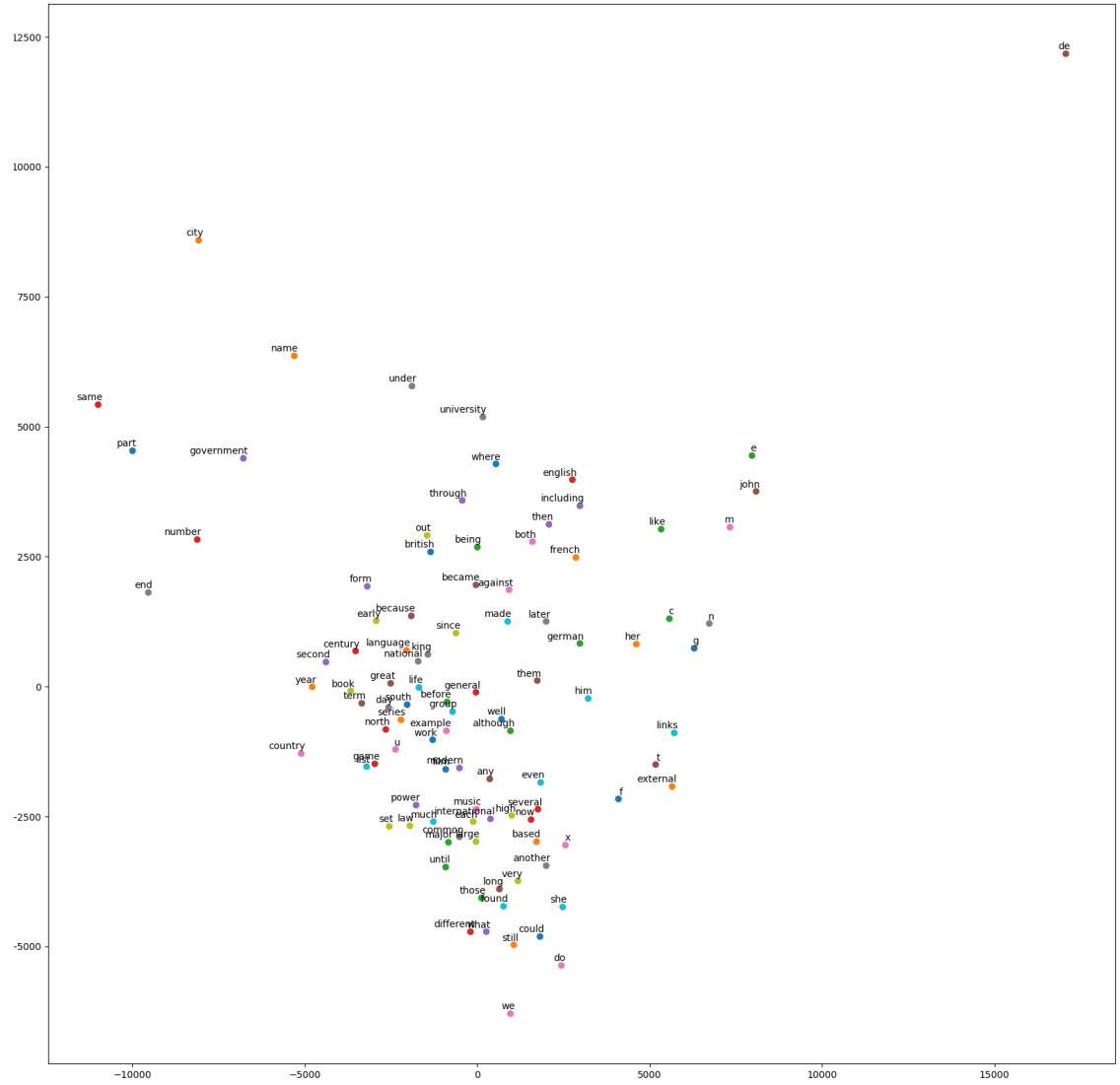


Figure 8: t-SNE projections of LSA Model

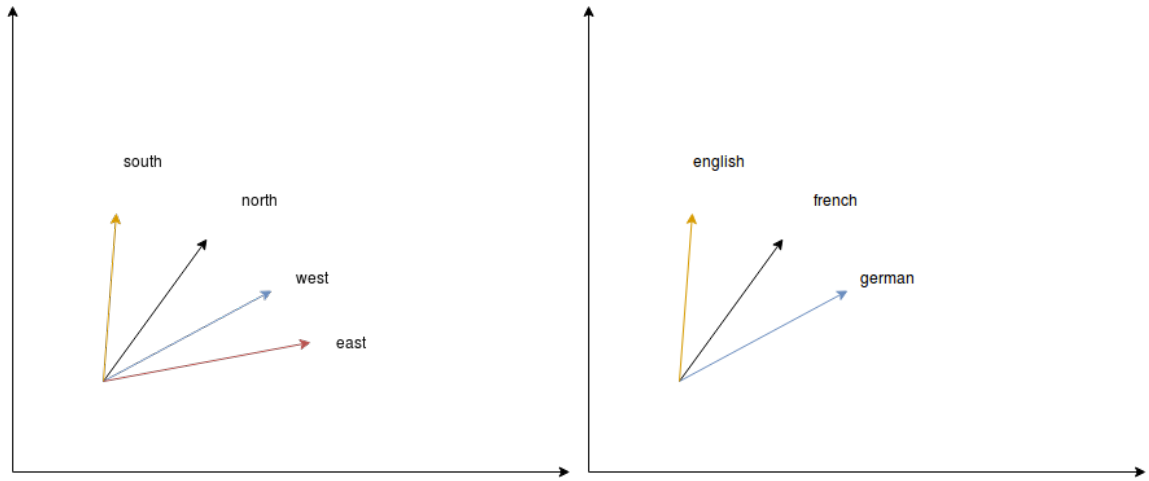
In the t-SNE projections we observe that words like 'modern','film' appear together and certain words like 'south', 'north' appear together. Words like 'major', 'large', 'long', 'very' also appear together

Here when using the LSA Model, we also find the closest embeddings to a given embedding. We observe the following patters.

Given a word, the closest word given are as follows:

1. city ['government', 'name', 'state']
2. english ['french', 'german', 'including']

3. them ['him', 'up', 'her']
4. since ['until', 'before', 'british']
5. century ['early', 'since', 'year']
6. north ['south', 'west', 'east']



(a) (b)

Figure 9: Words that have similar embeddings

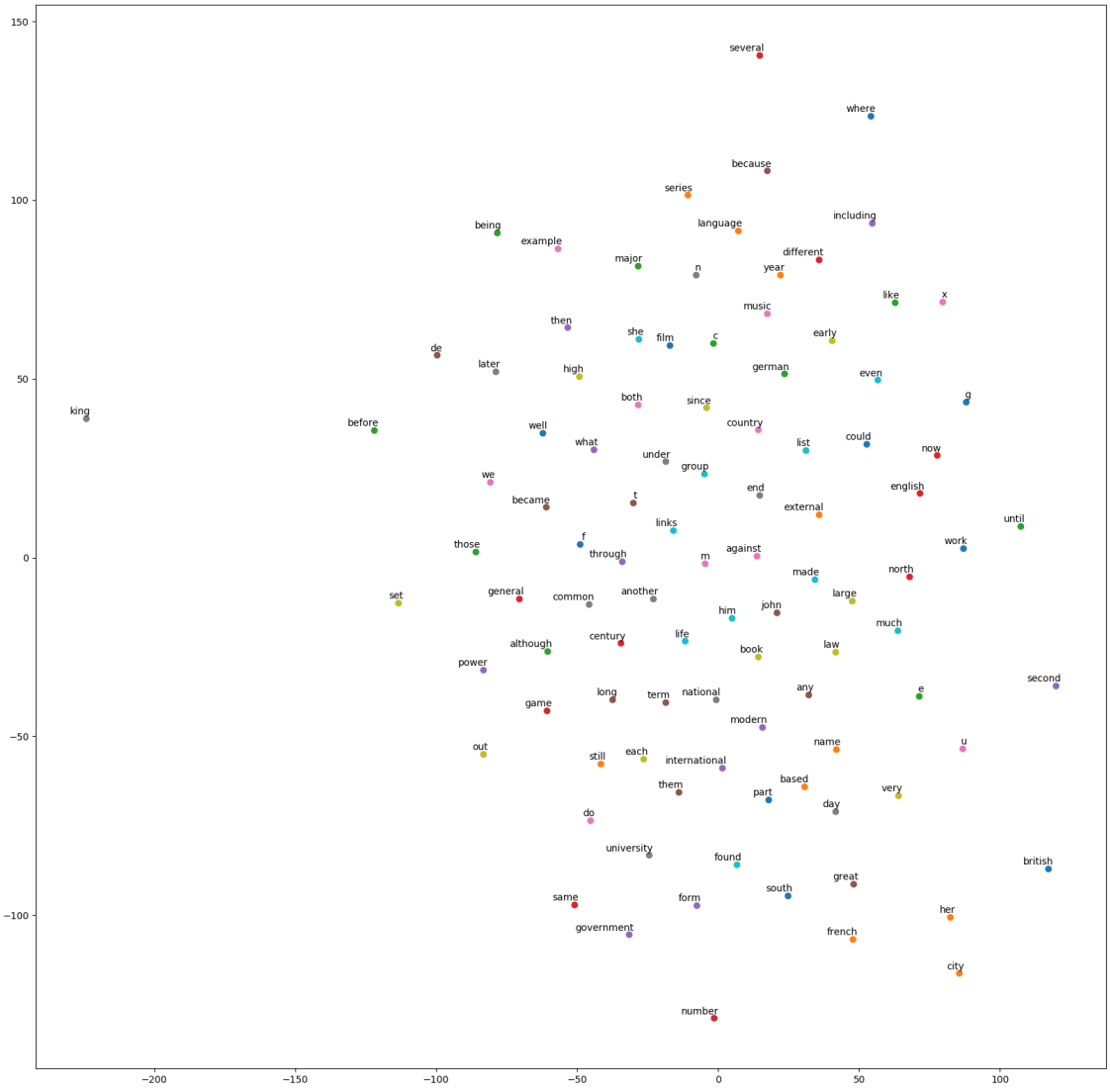


Figure 10: t-SNE projections of CBOW Model

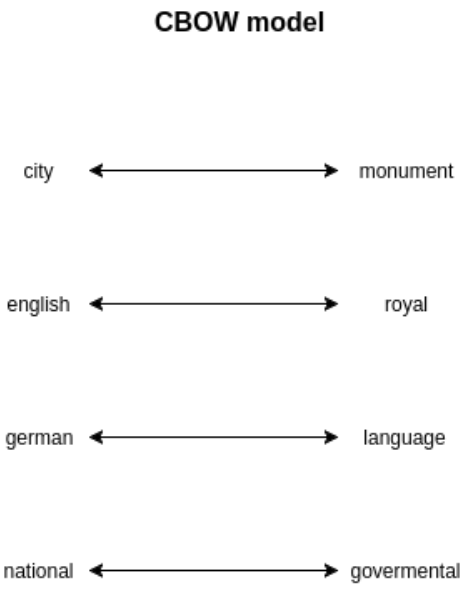


Figure 11: In the CBOW Model, we observe that the closest embeddings between pairs of words

4.2 Named Entity Recognition

This part was done by Desh Raj.

In this part, we were required to do an extrinsic evaluation of the word embeddings on a Named Entity Recognition (NER) task. The dataset to be used for this purpose was the CoNLL challenge dataset. However, since the embeddings were trained on the **text8** data, most of the words and entitites in the CoNLL dataset were not present in our trained word vector list. Due to this reason, the embeddings were basically useless for this task, since all the entities were being assigned the vector corresponding to **UNK**.

To solve this problem, we decided to perform the NER task on a sample of the **text8** data itself. Since the given **text8** data contains only lowercase words without punctuations or sentence segmentation, we downloaded the original "enwik8"[5] dataset from which the **text8** data has been created. "enwik8" contains the first 10^8 words of the English language Wikipedia articles. Since this dataset is in complex XML format, we used a Perl script (`convert.pl`) to extract the text from the data. Further, we used the POS tagger, chunk tagger, and NER tagger modules present in NLTK to convert this text into the same format as the CoNLL dataset (`extract.py`). Finally, around 950000 words were taken for training and 50000 for testing.

With these modifications, we expected to obtain decent results on the NER task, since now most of the words in the training corpus had a corresponding embedding trained earlier. However, the class distribution in the training and test sets were now highly skewed as follows.

Due to such a highly skewed distribution, the sliding window neural network does not perform well. In addition, we also tried a CNN-based classifier and an LSTM-based classifier, but none of these classifiers were able to perform satisfactorily on the dataset.

We argue that the failure of all of these classifiers is due to the distribution rather than the word embeddings. This is evident from the fact that even with additional features, namely POS tag, chunk

[5] http://mattmahoney.net/dc/textdata.html

Class Name	Training set	Test set
PERSON	37390 (3.94%)	913 (2.28%)
LOCATION	1100 (0.12%)	47 (0.18%)
ORGANIZATION	23591 (2.48%)	647 (1.62%)
GPE	26527 (2.79%)	892 (2.23%)
NONE	861392 (90.67%)	37500 (93.75%)

Table 1: Class distribution in training and test sets

tag, and first letter capital information, the networks could still not perform well. Cost-sensitive learning using a weighted objective function may be able to alleviate this problem, but we did not have sufficient time to evaluate such a hypothesis.