ASSIGNMENT 1A:

Word Embeddings with Word2Vec.

Neural Networks and Natural Language Processing

Carolina Raymond Rodrigues.
Student ID: 210607000
ec21209@qmul.ac.uk

Contents

1 Introduction **2**

2 Preprocessing the training corpus. **2**

3 Creating the corpus vocabulary and preparing the dataset. **3**

4 Building the skip-gram neural network architecture. **5**

5 Training the models. **7**
5.1 What would the inputs and outputs to the model be? 8
5.2 How would you use the Keras framework to create this architecture? 8
5.3 What are the reasons this training approach is considered inefficient? 9

6 Getting the word embeddings. **9**

7 Exploring and visualizing your word embeddings using t-SNE. **9**

List of Figures

1 Length of pre-processed output. 3
2 Created Corpus Vocabulary. 4
3 First sentence dataset preparation. 4
4 Context embeddings code. 5
5 Output layer code. 6
6 Compile model code. 6
7 Model summary. 7
8 Results of the training process. 7
9 Embedding vector of first 10 words. 9
10 Code for visualising the 5 most similar words in search terms. 10
11 Five most similar words for each search terms. 10
12 Visualisation of word embeddings using t-SNE. 10

List of Tables

1 Architecture framework. 8

1 Introduction

Embeddings are a dense vectorial representation used to encode the inputs of NLP models such as sentences or words. In this approach, the inputs with related meaning will be represented similarly.

The Word2Vec algorithm has the ability of training representations of a very large corpus. Therefore, it is used for building word embeddings and developing scalable models. Two architectures can be used for creating this embeddings: Continuous Bag of Words (CBOW) or Skip-Grams. In this part of the assignment, we are going to focus on the later one.

A Skip-Gram model uses a feedforward neural network with one hidden layer that learns how to predict the surrounding of given words. Nevertheless, the training process can be computationally expensive, since the number of weights our model has to learn will depend on the size of the vocabulary. To solve this inconvenience, negative sampling is applied, were only a small number of weights will be updated.

To perform the training, it has been decided to approach the problem as a binary classification, where the dataset will be structured in two columns. The first column represents the input and it's constituted with the target word and the context words. The size of the later one will depend on the window size. Moreover, the second column represents the label, in case the word is in the context of the target word, the value of this variable will be 1, otherwise it will be 0.

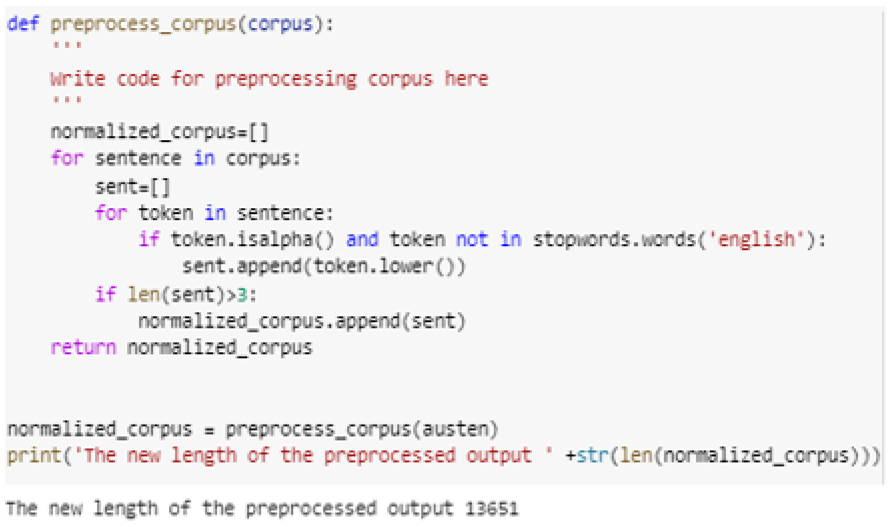
Taking into account all this information, we can now implement the script provided: "Lab 1: Skip-gram Model for Word2Vec".

2 Preprocessing the training corpus.

After downloading the training data from 3 documents of the Gutenberg corpus, the next step is pre-processing this data before implementing the Skip-Gram model. For that, characters are lowercased, punctuation marks, digits, sentences of length with fewer than 3 words and stop words are removed. This will help to get rid non-discriminative data or noise and consequently the performance of our model will improve..

This all will be done in the preprocess_corpus(corpus) function, whose input will be the corpus and the output will be a list of lists, where each inner list will contain the remaining tokens after the sentences have been pre-processed.

Once the pre-processing step has been implemented, a sanity check is performed in which it is expected that the total number of sentences of the three pre-processed Austen's corpus is around 13651. In the following figure, we can see that the pre-processing step is done correctly since the obtained output is 13651 sentences as expected.



```
def preprocess_corpus(corpus):
    '''
    Write code for preprocessing corpus here
    '''
    normalized_corpus=[]
    for sentence in corpus:
        sent=[]
        for token in sentence:
            if token.isalpha() and token not in stopwords.words('english'):
                sent.append(token.lower())
        if len(sent)>3:
            normalized_corpus.append(sent)
    return normalized_corpus


normalized_corpus = preprocess_corpus(austen)
print('The new length of the preprocessed output ' +str(len(normalized_corpus)))
```

```
The new length of the preprocessed output 13651
```

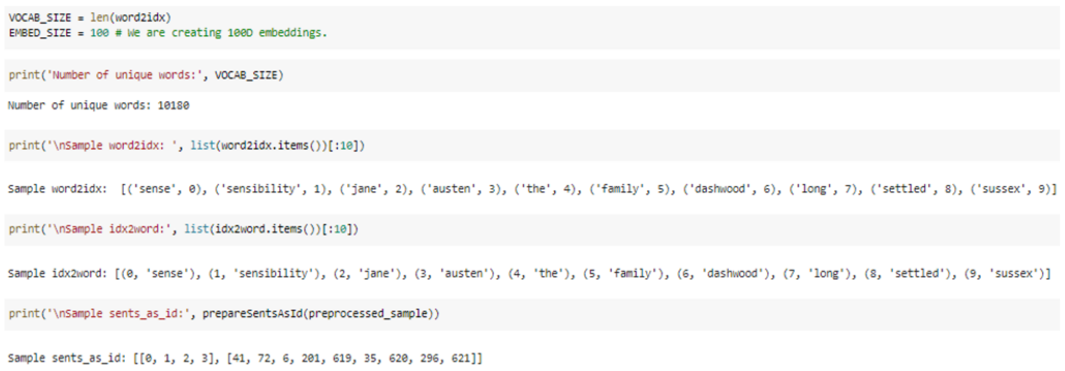
Figure 1: Length of pre-processed output.

3 Creating the corpus vocabulary and preparing the dataset.

Until now, we have a list of lists, whose inner lists are word tokens. Nevertheless, words cannot be feed to the Skip-Gram model. Therefore, an integer ID is assigned to each word and list of lists is build, in which every word token is replaced by its ID. Inside the function prepareData(normalised_corpus), two dictionaries will be implemented in order to fulfil our purpose. The first one, known as idx2word will be a dictionary whose key values will be unique integer indexes whose values correspond to the unique words. On the contrary, the second dictionary is word2idx, which is a reversal of the idx2word.

Once we have this two dictionaries, the word2idx dictionary will be used in order to set the sentences with word IDs instead of strings. This is done inside the prepareSentsAsId (normalized_corpus).

After implementing everything said in the previous paragraphs, sanity checks are performed in order to check if we have created correctly the corpus of the vocabulary. Firstly, it is expected that vocabulary length should be a value between 9800 and 12000. In our case we obtain a value of 10180. Secondly, we output the first 10 items of both dictionaries: word2idx and idx2word, to finally present how word sentences are transformed to ID sentences, using two sentences stored in the preprocessed_sample variable. As it can be seen in the following figure, the obtained output matches the output of the sanity check.



```
VOCAB_SIZE = len(word2idx)
EMBED_SIZE = 100 # We are creating 100D embeddings.
```

```
print('Number of unique words:', VOCAB_SIZE)
```

```
Number of unique words: 10180
```

```
print('\nSample word2idx: ', list(word2idx.items())[:10])
```

```
Sample word2idx:  [('sense', 0), ('sensibility', 1), ('jane', 2), ('austen', 3), ('the', 4), ('family', 5), ('dashwood', 6), ('long', 7), ('settled', 8), ('sussex', 9)]
```

```
print('\nSample idx2word:', list(idx2word.items())[:10])
```

```
Sample idx2word: [(0, 'sense'), (1, 'sensibility'), (2, 'jane'), (3, 'austen'), (4, 'the'), (5, 'family'), (6, 'dashwood'), (7, 'long'), (8, 'settled'), (9, 'sussex')]
```

```
print('\nSample sents_as_id:', prepareSentsAsId(preprocessed_sample))
```

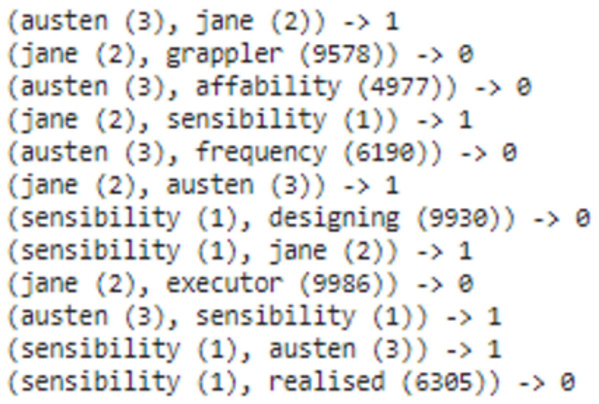
```
Sample sents_as_id: [[0, 1, 2, 3], [41, 72, 6, 201, 619, 35, 620, 296, 621]]
```

Figure 2: Created Corpus Vocabulary.

The next step is to prepare the dataset. As it was said in the introduction, the dataset will be structured in two columns. The first column represents the input and it's constituted with the target word and the context words, while the second column represents the label.

The function skipgrams that belongs to the tf.keras.preprocessing.sequence library will be in charge of doing this. This function takes as arguments each one of the sentences that belong to the normalised corpus transformed into word IDs, and the vocabulary size (previously estimated, whose value is 10180).

The Skip-Gram for the first sentence is visualised (Figure 3). The obtained results match the output provided in the sanity check.



```
(austen (3), jane (2)) -> 1
(jane (2), grappler (9578)) -> 0
(austen (3), affability (4977)) -> 0
(jane (2), sensibility (1)) -> 1
(austen (3), frequency (6190)) -> 0
(jane (2), austen (3)) -> 1
(sensibility (1), designing (9930)) -> 0
(sensibility (1), jane (2)) -> 1
(jane (2), executor (9986)) -> 0
(austen (3), sensibility (1)) -> 1
(sensibility (1), austen (3)) -> 1
(sensibility (1), realised (6305)) -> 0
```

Figure 3: First sentence dataset preparation.

As it can be seen in Figure 3, the skipgram function applies a window size of 2, where each word of the vocabulary and the context word (both represented by the word and it ID in parenthesis) are linked to a label with value 1 or 0 depending on if the word is in the context of the target word.

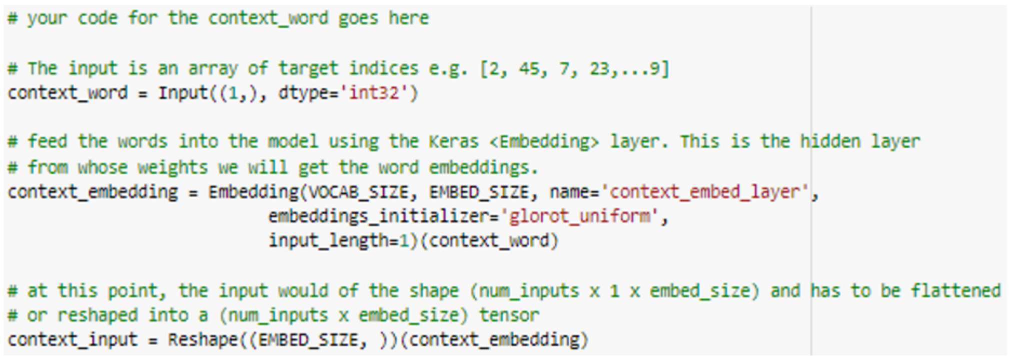
4 Building the skip-gram neural network architecture.

In this section, the skip-gram neural network architecture is built. As it was said in the introduction, the architecture of this model is quite simple: a feedforward neural network with one hidden layer. In this layer, the dot product between the input embeddings and context embeddings is computed. However, we must highlight that this layer does not use an activation function.

The result obtained from the hidden layer, will be the input of the output layer in which sigmoid activation function is used to compute the probabilities of word appearing to be the context of a given word.

All of this is done in 6 steps in the code:

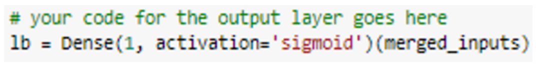
- Create input embeddings using the Keras Embedding layer library. The dimensions of these embeddings have been set to 100 and the weights initialization is done using the Glorot Uniform initializer, in which samples are drawn from a uniform distribution. Once the embedding is built, we must reshape it into (num_inputs, 100) tensor in order to perform the dot product.
- Create context embeddings following the same procedure as in the case of the input embeddings.



```
# your code for the context_word goes here

# The input is an array of target indices e.g. [2, 45, 7, 23,...9]
context_word = Input((1,), dtype='int32')

# feed the words into the model using the Keras <Embedding> layer. This is the hidden layer
# from whose weights we will get the word embeddings.
context_embedding = Embedding(VOCAB_SIZE, EMBED_SIZE, name='context_embed_layer',
                          embeddings_initializer='glorot_uniform',
                          input_length=1)(context_word)

# at this point, the input would of the shape (num_inputs x 1 x embed_size) and has to be flattened
# or reshaped into a (num_inputs x embed_size) tensor
context_input = Reshape((EMBED_SIZE, ))(context_embedding)
```

Figure 4: Context embeddings code.

- Subsequently, the dot product between the input and context embedding is performed using the Dot() function from Keras library. Computing the dot product will help with our purpose since it will learn the degree of closeness between the context and target word.
- The result of the dot product is a single value vector that will provide the cosine distance between the context and target word. This vector will be fed into the final layer whose activation function is the sigmoid. The output of this layer will provide the probability of the word appearing to be the context of a given word.



```
# your code for the output layer goes here
lb = Dense(1, activation='sigmoid')(merged_inputs)
```

Figure 5: Output layer code.

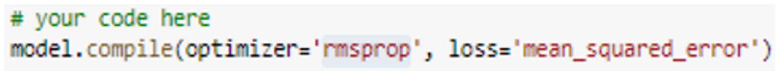
- Now that we have built the network, the next step is to initialize the model using the Keras Model() class, whose arguments are the input (list that will contain the inputs of the model, in this case the target and context word) and output (which will be the result of the output layer, calculated in the previous step).
- Finally, the model is configured for training using the compile method. Whose arguments are:

  o Loss function: measures how good is our model's prediction with respect to the ground truth value. In this case, the Mean Square Error (MSE) is used as the loss function, using Formula 1 as the estimator of this difference.

  $$MSE = \frac{1}{N}\sum_{i=1}^{M}(y_i - \hat{y_i})^2 \tag{1}$$

  Where N is the number of tested samples, y_i the model's prediction and $\hat{y}_i$ the ground truth value.

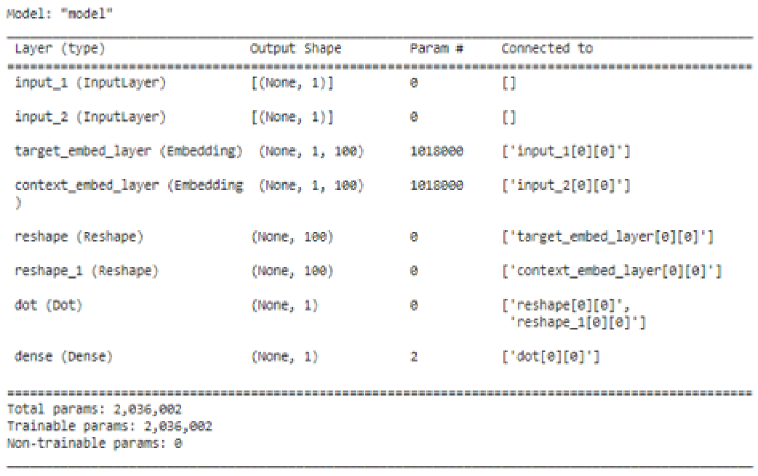
  o Optimizer: used to update the weights and biases in order to reduce the error between the prediction of the mod00el and the ground truth value. In this case the Root Means Squared Propagation (RMSProp) is utilized as optimizer.



```
# your code here
model.compile(optimizer='rmsprop', loss='mean_squared_error')
```

Figure 6: Compile model code.

At last, the model summary is printed. This will provide information of the number of each layers in order, their output shape and the number of parameters. Ultimately, the total number of parameters of the model will be shown as it can be seen in the following figure.



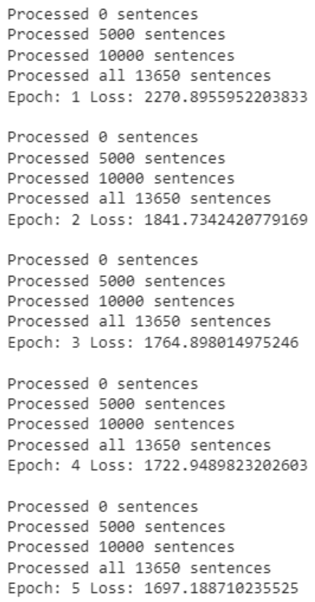
```
Model: "model"
_____________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
=============================================================================================
 input_1 (InputLayer)           [(None, 1)]          0           []

 input_2 (InputLayer)           [(None, 1)]          0           []

 target_embed_layer (Embedding)  (None, 1, 100)      1018000     ['input_1[0][0]']

 context_embed_layer (Embedding  (None, 1, 100)      1018000     ['input_2[0][0]']
 )

 reshape (Reshape)              (None, 100)          0           ['target_embed_layer[0][0]']

 reshape_1 (Reshape)            (None, 100)          0           ['context_embed_layer[0][0]']

 dot (Dot)                      (None, 1)            0           ['reshape[0][0]',
                                                                  'reshape_1[0][0]']

 dense (Dense)                  (None, 1)            2           ['dot[0][0]']

=============================================================================================
Total params: 2,036,002
Trainable params: 2,036,002
Non-trainable params: 0
_____________________________________________________________________________________________
```

Figure 7: Model summary.

5 Training the models.

In this section the training of the model is performed, where the parameters values (weights and biases) will be learned in order to make accurate predictions.

The model is trained in 5 epochs and the results obtained can be seen in the following figure, where it is observable that the algorithm slowly converges as the loss value reduces after each epoch.



```
Processed 0 sentences
Processed 5000 sentences
Processed 10000 sentences
Processed all 13650 sentences
Epoch: 1 Loss: 2270.8955952203833

Processed 0 sentences
Processed 5000 sentences
Processed 10000 sentences
Processed all 13650 sentences
Epoch: 2 Loss: 1841.7342420779169

Processed 0 sentences
Processed 5000 sentences
Processed 10000 sentences
Processed all 13650 sentences
Epoch: 3 Loss: 1764.898014975246

Processed 0 sentences
Processed 5000 sentences
Processed 10000 sentences
Processed all 13650 sentences
Epoch: 4 Loss: 1722.9489823202603

Processed 0 sentences
Processed 5000 sentences
Processed 10000 sentences
Processed all 13650 sentences
Epoch: 5 Loss: 1697.188710235525
```

Figure 8: Results of the training process.

In this section 3 questions must be answered:

5.1 What would the inputs and outputs to the model be?

The input of the model would be a set whose first item represents the target word and the second the context word. The size of the later one will depend on the window size, for instance, if the window size is 2, we will use one context word.

The input can't be feed as a text in our model. Hence, they will be represented as one-hot vectors. The dimensions of this vector will be equal to the vocabulary of words build from our training set. Thus, a 1 will be placed in the position that belongs to the word that will be analysed, and 0s in the other positions.

On the other hand, the output of the model will be a vector whose dimensions will be equal to the vocabulary size. Since the SoftMax function is used as the activation function in the final layer. Each one of the outputs will represent the probabilities of a word (defined by its position in the output vector) appearing to be the context of a given word. the output of each word will be a value between 0 and 1 and the sum of all probabilities should be 1.

5.2 How would you use the Keras framework to create this architecture?

Keras is an open-source software library that provides a Python interface for artificial neural networks.

Some of the Keras library implementations have been used to create the neural architecture used to fulfill the given task. They have been summarised in Table 1.

Error Function	Keras Module	Purpose
Skipgrams	preprocessing.sequence	Generates skipgram word pairs.Used for producing training instances.
Input	layers	To instantiate a Keras tensor. Used for building the model.
Dot	layers	Calculates the dot product between 2 tensors.Used for building the model.
Reshape	layers.core	Reshape the input to a given shape. Used for building the model.
Dense	layers.core	Adds a dense neural layer Used for building the model.
Embeddings	layers.embeddigns	Used to build the embedding layer.
Model	models	Creates the neural network model
plot_models	utils.vis_utils	Visualise a graph of the model

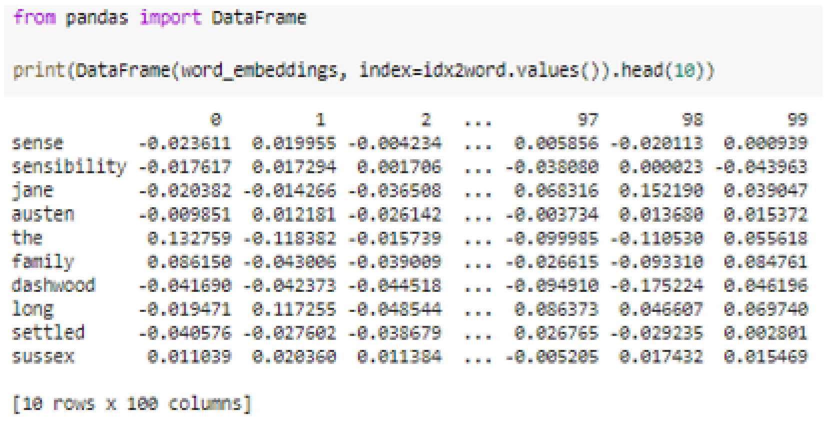
Table 1: Architecture framework.

5.3 What are the reasons this training approach is considered inefficient?

The main problem of this approach is the training process can be computationally very expensive. The number of parameters that the model has to learn will depend on the vocabulary size and the embedding size. As it can be seen in Figure 7, our model has to learn 2,036,002 parameters what makes this approach inefficient.

6 Getting the word embeddings.

We have set the embedding size to 100, therefore 100 features will be learned for every word in the vocabulary (10180 words). In this section the embeddings vector of the first 10 target words in the vocabulary are visualised. Where each value represents the weight of each feature.



```
from pandas import DataFrame

print(DataFrame(word_embeddings, index=idx2word.values()).head(10))
```

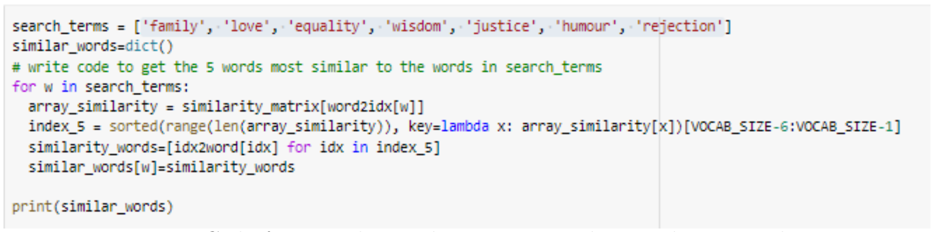
```
                    0         1         2  ...        97        98        99
sense       -0.023611  0.019955 -0.004234  ...  0.005856 -0.020113  0.000939
sensibility -0.017617  0.017294  0.001706  ... -0.038080  0.000023 -0.043963
jane        -0.020382 -0.014266 -0.036508  ...  0.068316  0.152190  0.039047
austen      -0.009851  0.012181 -0.026142  ... -0.003734  0.013680  0.015372
the          0.132759 -0.118382 -0.015739  ... -0.099985 -0.110530  0.055618
family       0.086150 -0.043006 -0.039009  ... -0.026615 -0.093310  0.084761
dashwood    -0.041690 -0.042373 -0.044518  ... -0.094910 -0.175224  0.046196
long        -0.019471  0.117255 -0.048544  ...  0.086373  0.046607  0.069740
settled     -0.040576 -0.027602 -0.038679  ...  0.026765 -0.029235  0.002801
sussex       0.011039  0.020360  0.011384  ... -0.005205  0.017432  0.015469

[10 rows x 100 columns]
```

Figure 9: Embedding vector of first 10 words.

7 Exploring and visualizing your word embeddings using t-SNE.

In this section, we will start computing the cosine similarity between the words in the vocabulary using the embedding layer. For that we will use the cosine_similarity() function of the sklearn library. The output of this function is a VOCAB_SIZExVOCAB_SIZE= 10180x10180 matrix, where each value represents the similarity among the words represented in the column and row of the matrix.

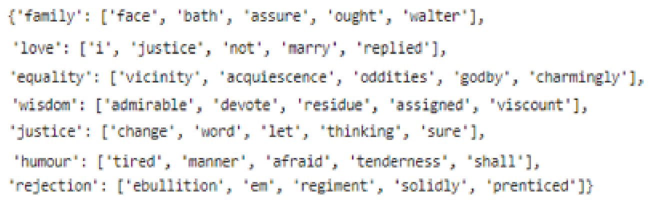
Taking this into account, we can print the 5 most similar words for the given search terms (’family’, ’love’, ’equality’, ’wisdom’, ’justice’, ’humour’, ’rejection’)



```
search_terms = ['family', 'love', 'equality', 'wisdom', 'justice', 'humour', 'rejection']
similar_words=dict()
# write code to get the 5 words most similar to the words in search_terms
for w in search_terms:
  array_similarity = similarity_matrix[word2idx[w]]
  index_5 = sorted(range(len(array_similarity)), key=lambda x: array_similarity[x])[VOCAB_SIZE-6:VOCAB_SIZE-1]
  similarity_words=[idx2word[idx] for idx in index_5]
  similar_words[w]=similarity_words

print(similar_words)
```

Figure 10: Code for visualising the 5 most similar words in search terms.

The output of this is:



```
{'family': ['face', 'bath', 'assure', 'ought', 'walter'],
 'love': ['i', 'justice', 'not', 'marry', 'replied'],
 'equality': ['vicinity', 'acquiescence', 'oddities', 'godby', 'charmingly'],
 'wisdom': ['admirable', 'devote', 'residue', 'assigned', 'viscount'],
 'justice': ['change', 'word', 'let', 'thinking', 'sure'],
 'humour': ['tired', 'manner', 'afraid', 'tenderness', 'shall'],
 'rejection': ['ebullition', 'em', 'regiment', 'solidly', 'prenticed']}
```

Figure 11: Five most similar words for each search terms.

Finally, 50-word embeddings are visualized using t-SNE function of the sklearn library. The output can be visualised in the following figure. Where the distance between the most similar words will be smaller. For instance the words "life" and "generations" are more alike than the words "life" and "supply".

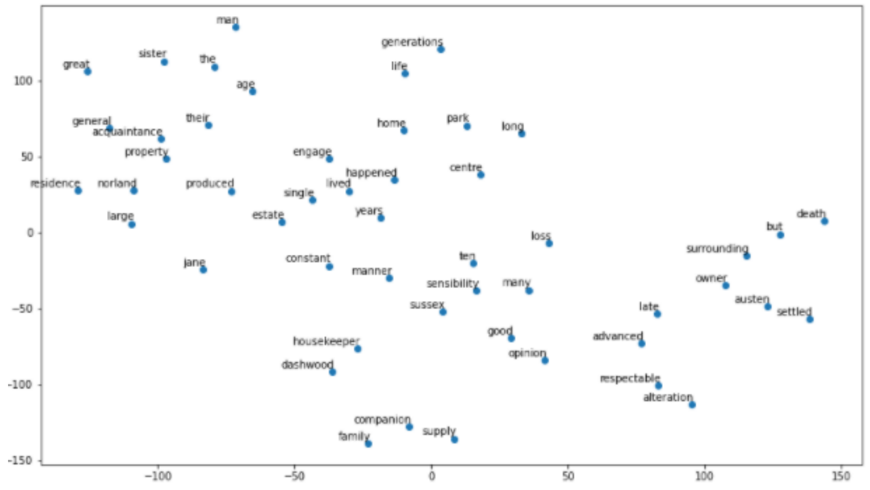


Figure 12: Visualisation of word embeddings using t-SNE.