Allan Roy Corinaldi Castaño - 201712677
Diego Andrés Gómez Polo - 201713198
Isabela María Sarmiento Llanos - 201731759
Yacir Andrés Ramirez Acevedo - 201114252

HW03 - Natural Language Processing

Simpsons Dialogues

1. Build Words Embeddings

- The data was already prepared as it had the format raw_character_text,spoken_words, where raw_character_text was the column that contained the character, and spoken_words the dialog of the character.
- The column spoken_words was processed using gensim.utils.simple_processed that handles lowercase letters and putting all individual words into an array without punctuations or other symbols. Every null row was removed from the dataframe and parse the list of processed dialogs into a numpy array.
- Gensim Word2Vec was used to generate embeddings of vector size 150, 100, and 50. The parameters used were a window of 10, min_count of 2 which ignores every word with frequency lower thatn 2.workers = 10, that are used as a threads to train the embeddings. And sg = 1, which means that the algorithm used was skip-gram. Finally, every model generated by those parameters were trained in order to convert a vector of words into embeddings where every word became a vector of size 150, 100, or 50 with floating values which means that we have a vector of vectors as embeddings.

The embeddings of different vector size will be available within the file.

2. Embedding Plotting

The n-dimensional embeddings are vectors that represent how similar semantically speaking are to words according to the context they were trained with. Since the embeddings had 50, 100 and 150 it is not possible to visualize the vectors in their corresponding vectorial space, however they can be transformed into bidimensional embeddings in order to plot them in a cartesian plane.

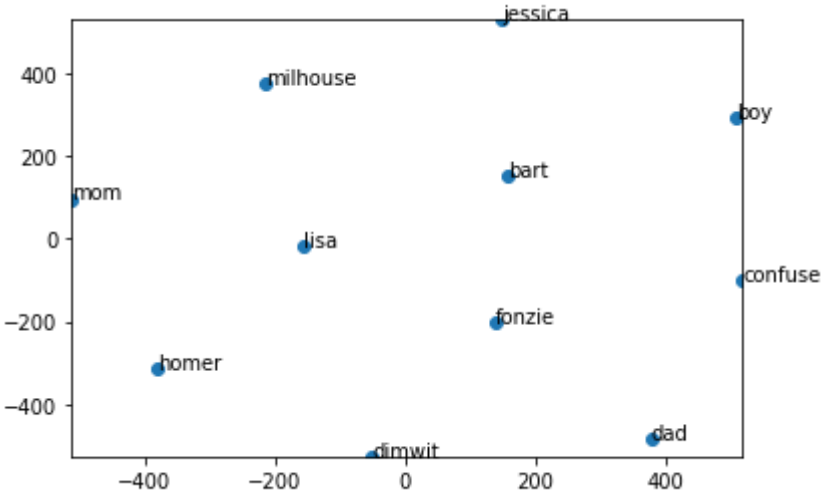
A common strategy used to plot embeddings in two dimensions is t-Distributed Stochastic Neighbor Embedding (t-SNE). This is a technique used for dimensionality reduction, which is “the process of reducing the number of random variables under consideration by obtaining a set of principal variables”[1]. The goal is to reduce the dimensionality of the embeddings in order to have fewer relationships between the features so that they can be explored and visualized with ease. This can be done either by feature elimination, selection or extraction.

[1] Introduction to t-SNE https://www.datacamp.com/community/tutorials/introduction-t-sne

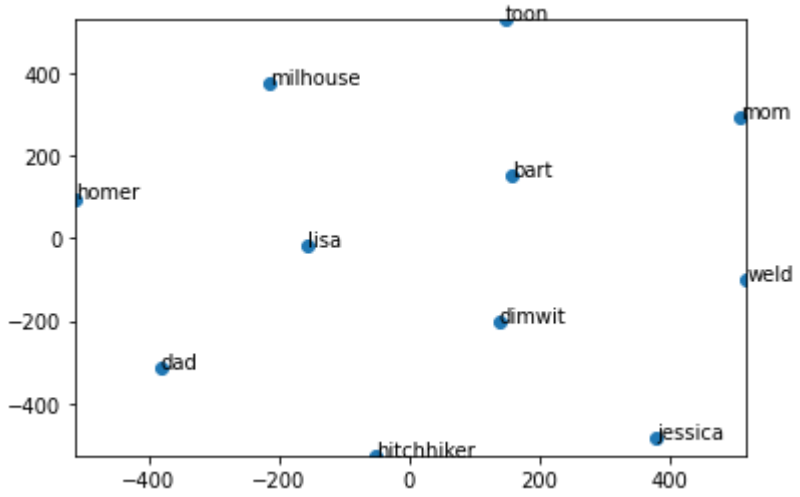
t-SNE calculates the conditional probability that two points are similar first in the original dimensional space and then in the reduced dimensional space. Then, the algorithm tries to minimize the difference between these probabilities by changing the representation of the small embeddings using the Kullbacl-Leibler divergence using the gradient descent method. This measure, the Kullbacl-Leibler divergence, reflects how a probability distribution diverges from the expected probability distribution. Therefore, t-SNE is able to find similarities between the embeddings and clustering them, and later visualize them in a bidimensional space for example.

In order to use t-SNE, we imported the TSNE method from sklearn and used Gensim methods such as similar_by_word(word) that returned the similar words when using the original embeddings. Using this method for each word and then calculating the corresponding coordinates for t-SNE, we were able to plot in a two dimensional plane the closest words for each character. The results obtained are shown in the following diagrams.

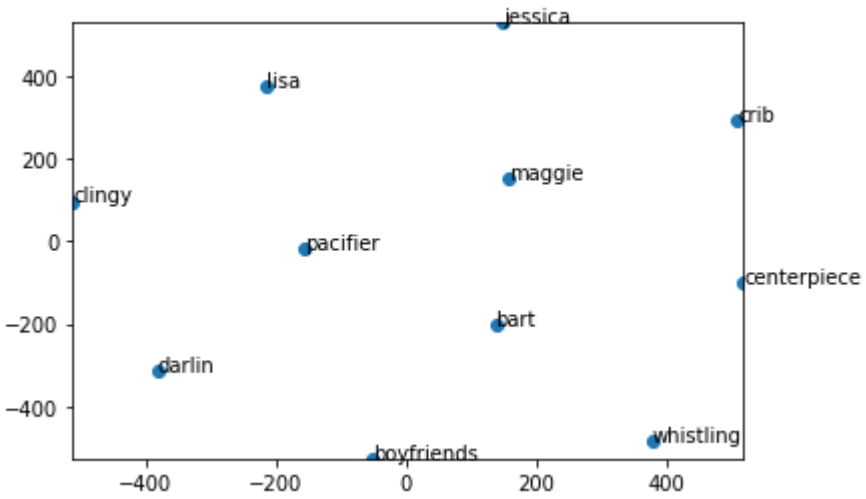
Similar Words to Bart (50):



Similar Words to Bart (100)

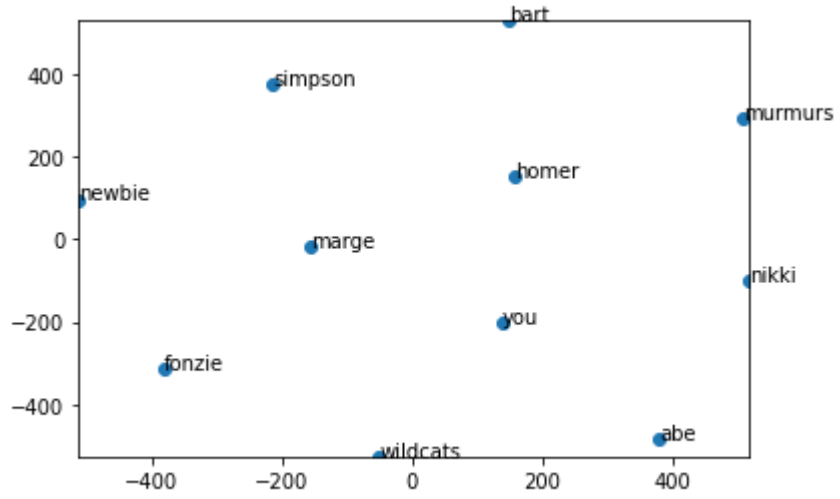


Similar Words to Bart (150)

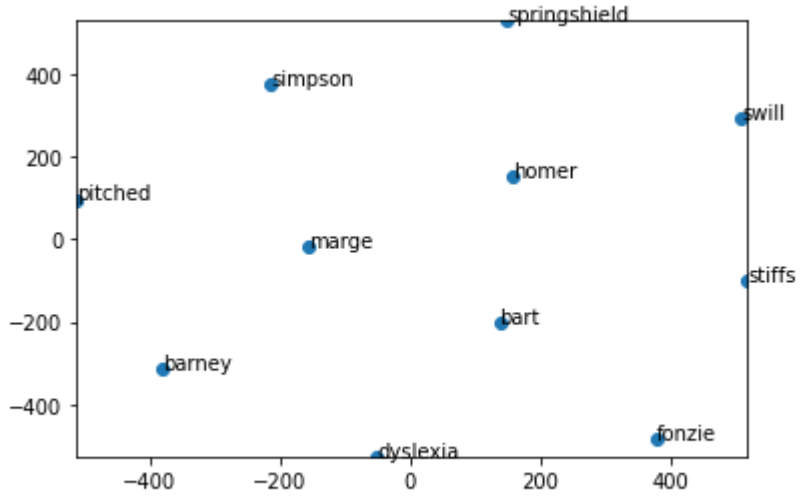


As the graphics from different embeddings show, the embeddings with 50 and 100 size, has almost the exactly same words but in different position. However, in the embeddings with size 150 the only word there are fewer words repeated such as “jessica” and “lisa”, that means different size of embeddings change words that can be similar among a word. It can happen as well that some words are so close that no matter the size of the embedding, it’s going to appear close such as “lisa” and “jessica”. This is most likely for the metric of the distance, the more distance between two words, the more more liley is to be more far away with other embeddings.

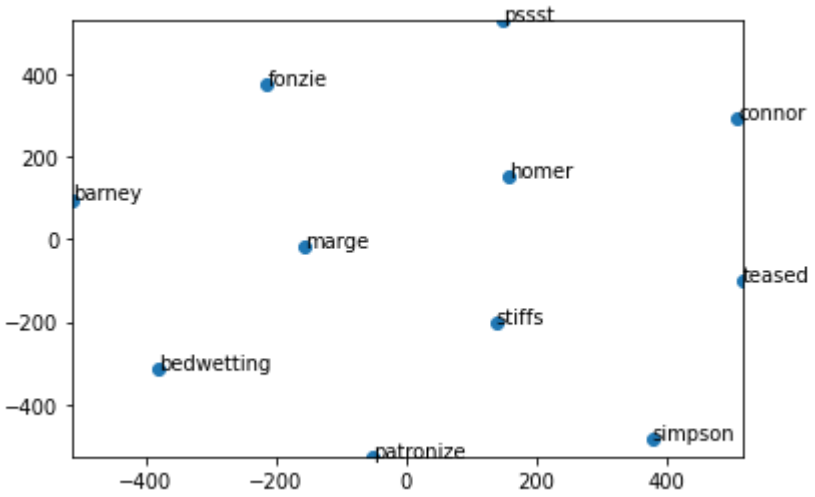
Similar Words to Homer(50)



Similar Words to Homer (100)



Similar Words to Homer (150)



In the images above with the closest words to homer, it's more clear that the closest words to homer in the diagram maintain the distance for embeddings of different size (example:marge), while the other words almost outside of the diagram are changing with the size of the embeddings.

3. Architectures with pre-trained embeddings

The preparation of the dataset was through dropping any null row, and filtering the dialogs with only main characters that were considered Lisa, Bart, Homer, Marge, C. Montgomery Burns, and Krusty the Clown into the dataframe. Those characters were considered by counting a group by the column raw_character_text, so the characters that more spoke were the main ones. A OneHotEncoder layer was used in order to map every row of raw_character_text from name of the character to a vector of size of main characters used (in this case 6) where every element of the vector was 0 except for the name of the character in that row which is 1. The idea of this OneHotEncoder is to have a vector representation of the category selected for the classifier. The dataset was distribuited in

testing 20%, and the remaining was distribuited in 20% validation and 60% training. Datasets splitted were converted in a tensor flow format.

Sample of the main characters

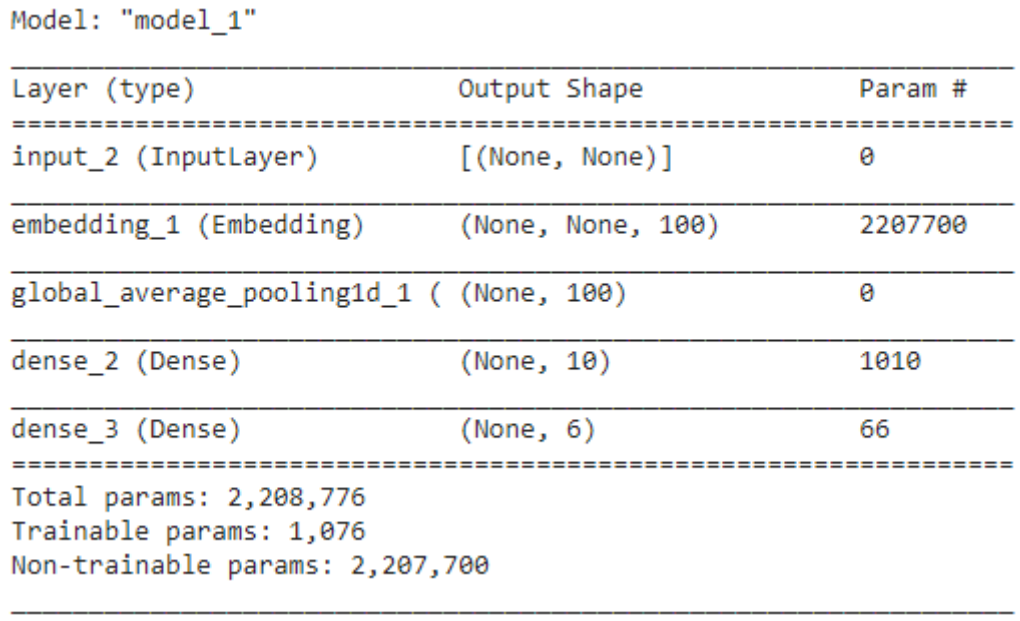
raw_character_text	
Homer Simpson	3381
Bart Simpson	1756
Marge Simpson	1393
Lisa Simpson	1003
C. Montgomery Burns	531
Krusty the Clown	303

There were two vocabularies, the one created by the pretrained embeddings that we saved and the ones generated by the dataset with vectorize_layer. So the embeddings were created with the interception between those vocabulary sets with the embeddings from the pretrainned ones. Once the weights (embeddings as inteception between the two vocabularies) are calculated we use the Embeddings layer in the three FeedForward architectures and passed as a parameter `embeddings_initializer` with the weights. The size of the vocabulary is 22077. The second layer used among all the architectures was Input that received int values that corresponds to the index of the word in the vocabulary. Finally, the Pooling layer was used to generate one feature map for each corresponding category of the classification task in the last layer instead of a dense layer.

First Architecture: A dense activation layer that had 10 cells and used the ‘relu’ algorithm as the activation function, and a dense layer with the softmax algorithm in order to give a distribution of probability for every label.

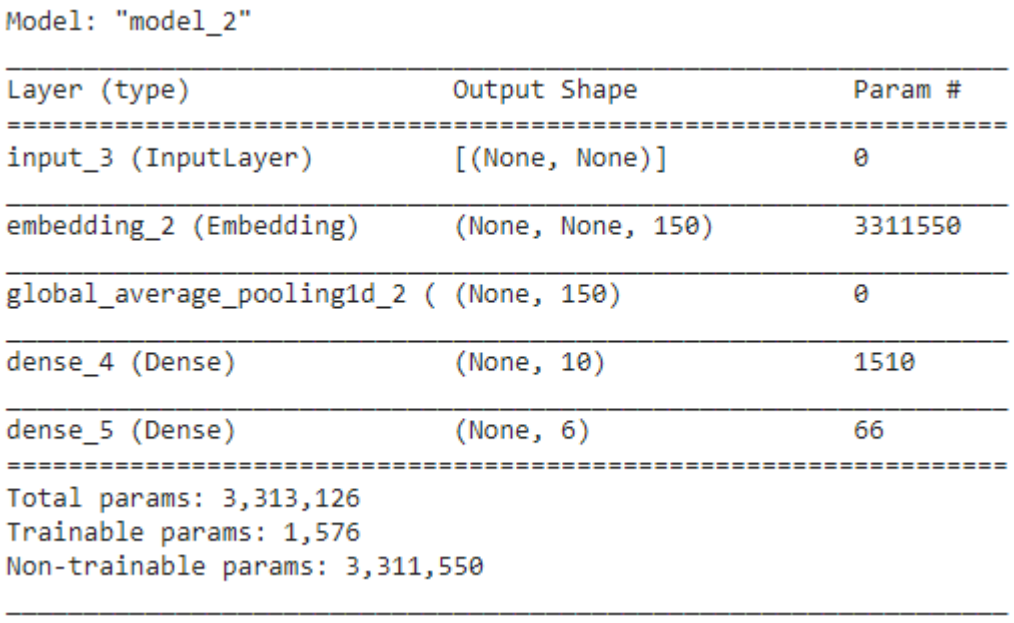
```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding (Embedding)        (None, None, 50)          1103850
_________________________________________________________________
global_average_pooling1d (Gl (None, 50)                0
_________________________________________________________________
dense (Dense)                (None, 10)                510
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 66
=================================================================
Total params: 1,104,426
Trainable params: 576
Non-trainable params: 1,103,850
_________________________________________________________________
```

In the image above the model with embeddings of size 50, and a vocabulary size of 22077.



```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_1 (Embedding)      (None, None, 100)         2207700
_________________________________________________________________
global_average_pooling1d_1 ( (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
_________________________________________________________________
dense_3 (Dense)              (None, 6)                 66
=================================================================
Total params: 2,208,776
Trainable params: 1,076
Non-trainable params: 2,207,700
_________________________________________________________________
```

In the image above the model with embeddings of size 100, and a vocabulary size of 22077.



```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_2 (Embedding)      (None, None, 150)         3311550
_________________________________________________________________
global_average_pooling1d_2 ( (None, 150)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1510
_________________________________________________________________
dense_5 (Dense)              (None, 6)                 66
=================================================================
Total params: 3,313,126
Trainable params: 1,576
Non-trainable params: 3,311,550
_________________________________________________________________
```

In the image above the model with embeddings of size 150, and a vocabulary size of 22077.

Second Architecture: In this architecture there are two Dense layers with 80 and 250 units with the algorithm relu, and and a dense layer with the softmax algorithm in order to give a distribution of probability for every label.

```
Model: "model_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_3 (Embedding)      (None, None, 50)          1103850
_________________________________________________________________
global_average_pooling1d_3 ( (None, 50)                0
_________________________________________________________________
dense_6 (Dense)              (None, 80)                4080
_________________________________________________________________
dense_7 (Dense)              (None, 250)               20250
_________________________________________________________________
dense_8 (Dense)              (None, 6)                 1506
=================================================================
Total params: 1,129,686
Trainable params: 25,836
Non-trainable params: 1,103,850
_________________________________________________________________
```

In the image above the model with embeddings of size 50, and a vocabulary size of 22077.

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_4 (Embedding)      (None, None, 100)         2207700
_________________________________________________________________
global_average_pooling1d_4 ( (None, 100)               0
_________________________________________________________________
dense_9 (Dense)              (None, 80)                8080
_________________________________________________________________
dense_10 (Dense)             (None, 250)               20250
_________________________________________________________________
dense_11 (Dense)             (None, 6)                 1506
=================================================================
Total params: 2,237,536
Trainable params: 29,836
Non-trainable params: 2,207,700
_________________________________________________________________
```

In the image above the model with embeddings of size 100, and a vocabulary size of 22077.

```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_5 (Embedding)      (None, None, 150)         3311550
_________________________________________________________________
global_average_pooling1d_5 ( (None, 150)               0
_________________________________________________________________
dense_12 (Dense)             (None, 80)                12080
_________________________________________________________________
dense_13 (Dense)             (None, 250)               20250
_________________________________________________________________
dense_14 (Dense)             (None, 6)                 1506
=================================================================
Total params: 3,345,386
Trainable params: 33,836
Non-trainable params: 3,311,550
_________________________________________________________________
```

In the image above the model with embeddings of size 150, and a vocabulary size of 22077.

Third Architecture: First, there is a Dense layers with 80, then a Dropout with a rate of 30% in order to avoid overfitting. Third, a batch normalization layer, a 250 units with the algorithm relu. fifth, another dropout layer with 30%, and a dense layer with the softmax algorithm in order to give a distribution of probability for every label.

```
Model: "model_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_6 (Embedding)      (None, None, 50)          1103850
_________________________________________________________________
global_average_pooling1d_6 ( (None, 50)                0
_________________________________________________________________
dense_15 (Dense)             (None, 80)                4080
_________________________________________________________________
dropout (Dropout)            (None, 80)                0
_________________________________________________________________
batch_normalization (BatchNo (None, 80)                320
_________________________________________________________________
dense_16 (Dense)             (None, 250)               20250
_________________________________________________________________
dropout_1 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_17 (Dense)             (None, 6)                 1506
=================================================================
Total params: 1,130,006
Trainable params: 25,996
Non-trainable params: 1,104,010
_________________________________________________________________
```

In the image above the model with embeddings of size 50, and a vocabulary size of 22077.

```
Model: "model_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_7 (Embedding)      (None, None, 100)         2207700
_________________________________________________________________
global_average_pooling1d_7 ( (None, 100)               0
_________________________________________________________________
dense_18 (Dense)             (None, 80)                8080
_________________________________________________________________
dropout_2 (Dropout)          (None, 80)                0
_________________________________________________________________
batch_normalization_1 (Batch (None, 80)                320
_________________________________________________________________
dense_19 (Dense)             (None, 250)               20250
_________________________________________________________________
dropout_3 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_20 (Dense)             (None, 6)                 1506
=================================================================
Total params: 2,237,856
Trainable params: 29,996
Non-trainable params: 2,207,860
_________________________________________________________________
```

In the image above the model with embeddings of size 100, and a vocabulary size of 22077.

```
Model: "model_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_8 (Embedding)      (None, None, 150)         3311550
_________________________________________________________________
global_average_pooling1d_8 ( (None, 150)               0
_________________________________________________________________
dense_21 (Dense)             (None, 80)                12080
_________________________________________________________________
dropout_4 (Dropout)          (None, 80)                0
_________________________________________________________________
batch_normalization_2 (Batch (None, 80)                320
_________________________________________________________________
dense_22 (Dense)             (None, 250)               20250
_________________________________________________________________
dropout_5 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_23 (Dense)             (None, 6)                 1506
=================================================================
Total params: 3,345,706
Trainable params: 33,996
Non-trainable params: 3,311,710
_________________________________________________________________
```

In the image above the model with embeddings of size 150, and a vocabulary size of 22077.

Metrics: For the 3 models in every architecture the results of the metrics were the same. Evrey prediction of the models was the label 2 that corresponds to Homer Simpsons, which is the character with most dialogs. The precision, recall and f1-score for the others labels was 0 except for label 2 that has a value between 0.4 and 0.39 precision, recall 1, and 0.57 or 0.56 in f1-score. The accuracy was between 0.4 and 0.39 for the model. This result was for validation, testing, and training. The results were found by predicting the specific the dialogs values of dataset (example x_val), then transforming those results from an array of probabilities to the index of the value with the biggest probability that corresponds to a class. After that, the y_val or the labeled data was mapped from the OneHotEncoder vector to take the only one and assign the index of that one. After having labels and predicted labels as classes we could find confusion matrix and metrics with `classification_report` from skitlearn that gives the precision, recall, and f1-score for every label.

	precision	recall	f1-score	support
0	0.00	0.00	0.00	340
1	0.00	0.00	0.00	107
2	0.40	1.00	0.57	675
3	0.00	0.00	0.00	64
4	0.00	0.00	0.00	206
5	0.00	0.00	0.00	282
accuracy			0.40	1674

	precision	recall	f1-score	support
0	0.00	0.00	0.00	2631
1	0.00	0.00	0.00	598
2	0.39	1.00	0.56	5510
3	0.00	0.00	0.00	2120
4	0.00	0.00	0.00	2666
5	0.00	0.00	0.00	607
accuracy			0.39	14132

The group speculated that these results were overfitting due to wrong partition of the data into the different datasets without taking care of the each label distributed proportionally.

4. Architectures with training layer

The first 2 Layers for all the Architecture where basically identical, a TextVectorization layer and an EmbeddingLayer

Text Vectorization:

Layer (type)	Output Shape	Param #
text_vectorization_1 (TextVe	(None, 20742)	0

This is the first Layer of the model. It is in charge of processing the raw train datasets. This will output a vector representation for each dialog of the dataset, these vectors will correspond to either Bow, int or tfidf representations for each dialog, the size of these representations will in general correspond to the length of our vocabulary (20742).. . The shape (None, None) represents a matrix of size batchSize x LengthDialog. This dimension clearly will be updated when we fit our model.

Embeddings:

embedding_Trainable (Embeddi	(None, 20742, 100)	2074200

The second layer will be in charge of training the embedding matrix for our model. In this case the shape (None, 20742, 100) is telling us that our generated embeddings will have a dimensionality of 100. The output number of params (2074200) corresponds to the number of cells in the output matrix of this layer which will be of shape (20742 x 100), basically, each

row represents the embedding for the corresponding token. Only tokens which appear in one dialog will have corresponding non-zero rows in the outputs matrix of this layer.

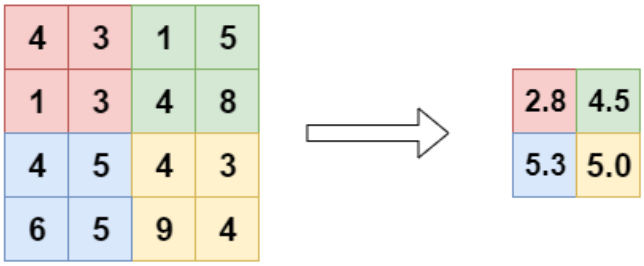
First Architecture:

Now for the first Architecture, apart from the previous mencioned layers, we choose a preatty conservative approach with the following Layers:

- TextVectorization
- Embeddings
- GlobalAveragePooling1D
- Dense(relu) with 10 nodes
- Output Dense(softmax) with 6 nodes, 1 per class

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
text_vectorization_1 (TextVe (None, 20742)             0
_________________________________________________________________
embedding_Trainable (Embeddi (None, 20742, 100)        2074200
_________________________________________________________________
global_average_pooling1d (Gl (None, 100)               0
_________________________________________________________________
dense (Dense)                (None, 10)                1010
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 66
=================================================================
Total params: 2,075,276
Trainable params: 2,075,276
Non-trainable params: 0
```

GlobalAveragePooling1D: As was said before, the embedding layer outputs a matrix, but the dense layers usually only accept 1D vectors, in that sense we used a GlobalAvergePooling1D layer to redimentionate our output to a one dimension vector suitable for the following layer (dense). This Layer creates a 1D output of a given length based on averages taken from the original input. The way it takes these averages is using “windows” of a fixed size and getting averages of the cells on those windows. Below is an example of a 2D Average Pooling process.



In our case the averages are taken Column-wise in the 20742x100 matrix so we end up with a 100x1 shaped vector.
Afterwards, the dense had 10 nodes, so the dimensions of the weights matrix of this layer would be (#nodes x (inputSize+1)). Where each row of this matrix represents the individual weights each node is giving to the input features, plus the weight given to de bias. InputSize corresponds to the size of the vector output by GlobalAveragePooling1d (100).
So (10 x (100+1)) = # of params = 1010. we must keep in mind that dense layers pass each row of the above matrix to an activation function, relu in this case, so they output a vector of shape 1x#rowsOfPreviousMatrix. Here it would output a column vector of size 10
.
Similarly, the last step also was a dense layer, the only difference, apart from the number of nodes (6, one for each target character), was the activation function, a softmax, used to convert our weights to probabilities. The same formula as before applies:
(#nodes x (inputSize+1)) = (6x (10 + 1)) = # of params = 66. With a 6 X 11 matrix. Passing each row vector through the softmax and we obtain the estimated probabilities of each target class.

Second Architecture:
The general design of this architecture is as folllows:

- TextVectorization
- Embeddings
- GlobalAveragePooling1D
- Dense(relu) with 80 nodes
- Dense(relu) with 250 nodes
- Output Dense(softmax) with 6 nodes, 1 per class

```
Model: "sequential_26"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
text_vectorization_10 (TextV (None, 20742)             0
_________________________________________________________________
embedding_Trainable (Embeddi (None, 20742, 100)        2074200
_________________________________________________________________
global_average_pooling1d_23  (None, 100)               0
_________________________________________________________________
dense_63 (Dense)             (None, 80)                8080
_________________________________________________________________
dense_64 (Dense)             (None, 250)               20250
_________________________________________________________________
dense_65 (Dense)             (None, 6)                 1506
=================================================================
Total params: 2,104,036
Trainable params: 2,104,036
Non-trainable params: 0
_________________________________________________________________
```

We won't go in too much detail on the specifics of this architecture because it is quite similar to the previous one and, in general, the same formulas apply. The particularity with this one is that we added more dense nodes in the middle.

Third Architecture:
The general design of this architecture is as folllows:

- TextVectorization
- Embeddings
- GlobalAveragePooling1D
- Dense(relu) with 80 nodes
- Dropout. 30%
- BatchNormalization
- Dense(relu) with 250 nodes
- Dropout. 50%
- Output Dense(softmax) with 6 nodes, 1 per class

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
text_vectorization_10 (TextV (None, 20742)             0
_________________________________________________________________
embedding_Trainable (Embeddi (None, 20742, 100)        2074200
_________________________________________________________________
global_average_pooling1d_26  (None, 100)               0
_________________________________________________________________
dense_72 (Dense)             (None, 80)                8080
_________________________________________________________________
dropout (Dropout)            (None, 80)                0
_________________________________________________________________
batch_normalization (BatchNo (None, 80)                320
_________________________________________________________________
dense_73 (Dense)             (None, 250)               20250
_________________________________________________________________
dropout_1 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_74 (Dense)             (None, 6)                 1506
=================================================================
Total params: 2,104,356
Trainable params: 2,104,196
Non-trainable params: 160
```

This architecture has quite a few tweaks in relations to the previous ones. It starts the same way with the usual TexVectorization, Embeddings, GlobalAvergaPoolin1D and Dense Layer. Next we make use of a new Layer, the Dropout one, which will ignore a fixed portion of the previous nodes. The first of these layers will ignore 30% of its previous nodes. The number of trainable params in this layer are 0. The we add other new Layer, a BatchNormalization Layer, its main purpose is to apply a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

5. Comparison

- **Evaluation with inplace Embeddings**: In general we saw much worse results using binary and tf-idf output modes for the TextVectorization Layer in all of the architectures used. The **int** output mode outperformed the two prior both in evaluation metrics and in training time. This might be because the EmbeddingLayer of Keras is mainly designed for getting **int** indexes as inputs, so this is how it performs the best.
    - 1st Architecture:

Architecture 1 binary Embedding

	precision	recall	f1-score	support
0	0.00	0.00	0.00	2621
1	0.00	0.00	0.00	381
2	0.42	1.00	0.59	5533
3	0.00	0.00	0.00	2145
4	0.00	0.00	0.00	3
5	0.00	0.00	0.00	2635
accuracy			0.42	13318
macro avg	0.07	0.17	0.10	13318
weighted avg	0.17	0.42	0.24	13318

Architecture 1 int Embedding

	precision	recall	f1-score	support
0	0.41	0.27	0.32	2621
1	0.29	0.08	0.13	381
2	0.54	0.64	0.59	5533
3	0.36	0.30	0.33	2145
4	0.00	0.00	0.00	3
5	0.39	0.47	0.43	2635
accuracy			0.46	13318
macro avg	0.33	0.29	0.30	13318
weighted avg	0.45	0.46	0.45	13318

Architecture 1 tfidf Embedding

	precision	recall	f1-score	support
0	0.00	0.00	0.00	2621
1	0.00	0.00	0.00	381
2	0.42	1.00	0.59	5533
3	0.00	0.00	0.00	2145
4	0.00	0.00	0.00	3
5	0.00	0.00	0.00	2635
accuracy			0.42	13318
macro avg	0.07	0.17	0.10	13318
weighted avg	0.17	0.42	0.24	13318

- 2do Architecture:

Architecture 2 binary Embedding

	precision	recall	f1-score	support
0	0.00	0.00	0.00	2621
1	0.00	0.00	0.00	381
2	0.42	1.00	0.59	5533
3	0.00	0.00	0.00	2145
4	0.00	0.00	0.00	3
5	0.00	0.00	0.00	2635
accuracy			0.42	13318
macro avg	0.07	0.17	0.10	13318
weighted avg	0.17	0.42	0.24	13318

Architecture 2 int Embedding

	precision	recall	f1-score	support
0	0.36	0.22	0.28	2621
1	0.23	0.11	0.15	381
2	0.55	0.56	0.55	5533
3	0.31	0.38	0.34	2145
4	0.00	0.00	0.00	3
5	0.38	0.46	0.41	2635
accuracy			0.43	13318
macro avg	0.30	0.29	0.29	13318
weighted avg	0.43	0.43	0.42	13318

Architecture 2 tfidf Embedding

	precision	recall	f1-score	support
0	0.00	0.00	0.00	2621
1	0.00	0.00	0.00	381
2	0.42	1.00	0.59	5533
3	0.00	0.00	0.00	2145
4	0.00	0.00	0.00	3
5	0.00	0.00	0.00	2635
accuracy			0.42	13318
macro avg	0.07	0.17	0.10	13318
weighted avg	0.17	0.42	0.24	13318

- 3rd Architecture:

Architecture 3 binary Embedding

	precision	recall	f1-score	support
0	0.00	0.00	0.00	2621
1	0.00	0.00	0.00	381
2	0.42	1.00	0.59	5533
3	0.00	0.00	0.00	2145
4	0.00	0.00	0.00	3
5	0.00	0.00	0.00	2635
accuracy			0.42	13318
macro avg	0.07	0.17	0.10	13318
weighted avg	0.17	0.42	0.24	13318

Architecture 3 int Embedding

	precision	recall	f1-score	support
0	0.33	0.38	0.35	2621
1	0.12	0.17	0.14	381
2	0.50	0.70	0.58	5533
3	0.37	0.14	0.21	2145
4	0.00	0.00	0.00	3
5	0.51	0.20	0.29	2635
accuracy			0.44	13318
macro avg	0.30	0.27	0.26	13318
weighted avg	0.44	0.44	0.41	13318

Architecture 3 tfidf Embedding

	precision	recall	f1-score	support
0	0.00	0.00	0.00	2621
1	0.00	0.00	0.00	381
2	0.42	1.00	0.59	5533
3	0.00	0.00	0.00	2145
4	0.00	0.00	0.00	3
5	0.00	0.00	0.00	2635
accuracy			0.42	13318
macro avg	0.07	0.17	0.10	13318
weighted avg	0.17	0.42	0.24	13318

- **Architectures with pretrain embeddings for testing and validation with 50, 100, and 150 vector size**

	precision	recall	f1-score	support
0	0.00	0.00	0.00	2631
1	0.00	0.00	0.00	598
2	0.39	1.00	0.56	5510
3	0.00	0.00	0.00	2120
4	0.00	0.00	0.00	2666
5	0.00	0.00	0.00	607
accuracy			0.39	14132

- **Architectures with pretrain embeddings for training with 50, 100, and 150 vector size**

	precision	recall	f1-score	support
0	0.00	0.00	0.00	340
1	0.00	0.00	0.00	107
2	0.40	1.00	0.57	675
3	0.00	0.00	0.00	64
4	0.00	0.00	0.00	206
5	0.00	0.00	0.00	282
accuracy			0.40	1674

As we can see, if we take the best metrics for inplace embeddings (**int** as output mode of the TextVectorizationLayer) the accuracy results between the two approaches, with and without pre trained embeddings, are quite similar, and actually a little bit better for the inplace embeddings (up to **0.45**). So, at least in this case, the idea of transferable knowledge with the use of pre trained embeddings is not worth the massive effort it takes to train embeddings on a large corpus.

Friends Dialogues

1. Build Words Embeddings

The dataset consisted of a folder of 228 .txt files, where each file contained the dialogues and comments for each episode. Initially, we processed these files with a Python script, in order to prepare the data considering only the dialogues, removing the comments and the empty lines. This was achieved using regular expressions that detected the dialogues that corresponded to the main characters. Also, the 228 files were merged into a single .csv file, separating the character and its corresponding dialogue. This allowed us to replicate the preprocessing that was done with the Friends dataset using Pandas and Gensim. The columns of the csv files were named in the same way as they were named in the Simpsons Dialogue, raw_character_tex and spoken_words.

The preprocessing was done using again gensim.utils.simple_processed, which lowered cased words, discarded random symbols and punctuation and split the complete strings of the dialogues into arrays that were later parsed to numpy arrays.

Using gensim.models.Word2Vec(), we were able to create the Word2Vect embeddings. The parameters used were the same as the Simpsons exercise, hence we created embeddings of dimension 50, 100 and 150. They were saved to disk and their corresponding file can be found on the attached folder.

2. Embedding Plotting

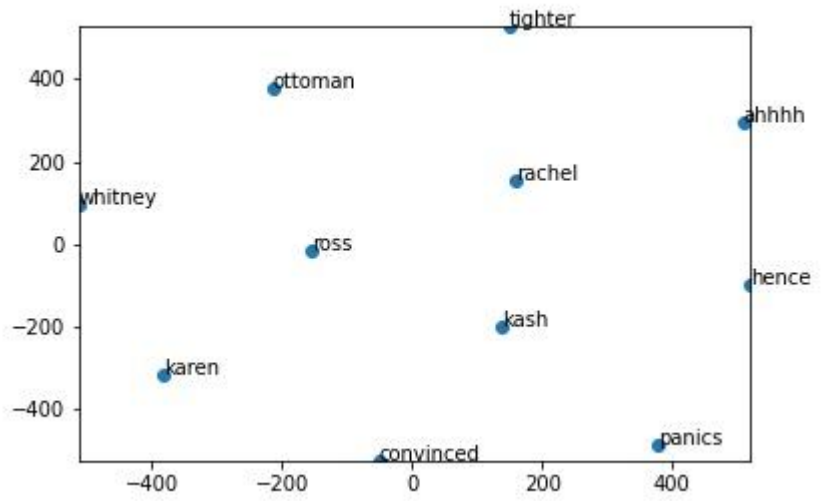
The n-dimensional embeddings are vectors that represent how similar semantically speaking are to words according to the context they were trained with. Since the embeddings had 50, 100 and 150 it is not possible to visualize the vectors in their corresponding vectorial space, however they can be transformed into bidimensional embeddings in order to plot them in a cartesian plane.

A common strategy used to plot embeddings in two dimensions is t-Distributed Stochastic Neighbor Embedding (t-SNE). This is a technique used for dimensionality reduction, which is "the process of reducing the number of random variables under consideration by obtaining a set of principal variables"[2]. The goal is to reduce the dimensionality of the embeddings in order to have fewer relationships between the features so that they can be explored and visualized with ease. This can be done either by feature elimination, selection or extraction. t-SNE calculates the conditional probability that two points are similar first in the original dimensional space and then in the reduced dimensional space. Then, the algorithm tries to minimize the difference between these probabilities by changing the representation of the small embeddings using the Kullbacl-Leibler divergence using the gradient descent method. This measure, the Kullbacl-Leibler divergence, reflects how a probability distribution diverges from the expected probability distribution. Therefore, t-SNE is able to find similarities between the embeddings and clustering them, and later visualize them in a bidimensional space for example.

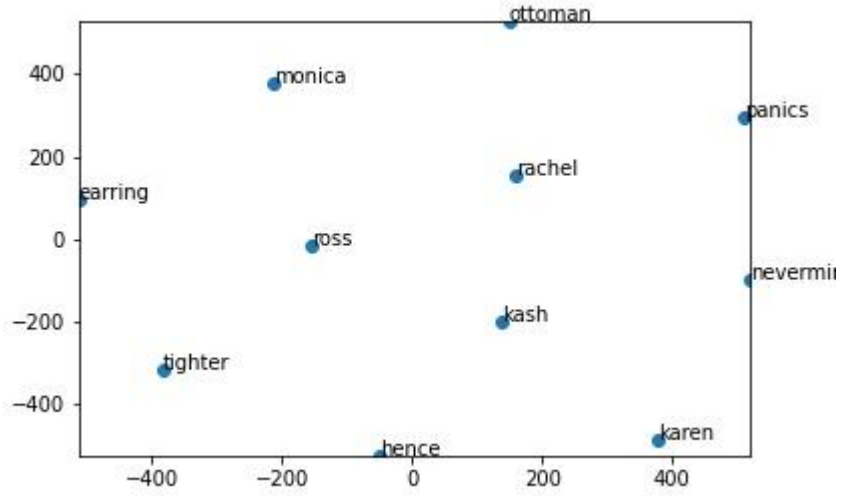
In order to use t-SNE, we imported the TSNE method from sklearn and used Gensim methods such as similar_by_word(word) that returned the similar words when using the original embeddings. Using this method for each word and then calculating the corresponding coordinates for t-SNE, we were able to plot in a two dimensional plane the closest words for each character. The results obtained are shown in the following diagrams.

Similar Words to Rachel (150 embeddings):

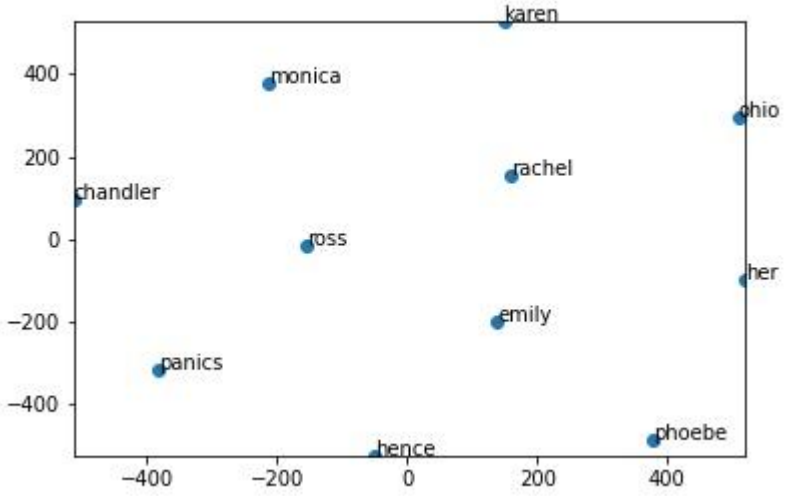
[2] Introduction to t-SNE https://www.datacamp.com/community/tutorials/introduction-t-sne



Similar Words to Rachel (100 embeddings):

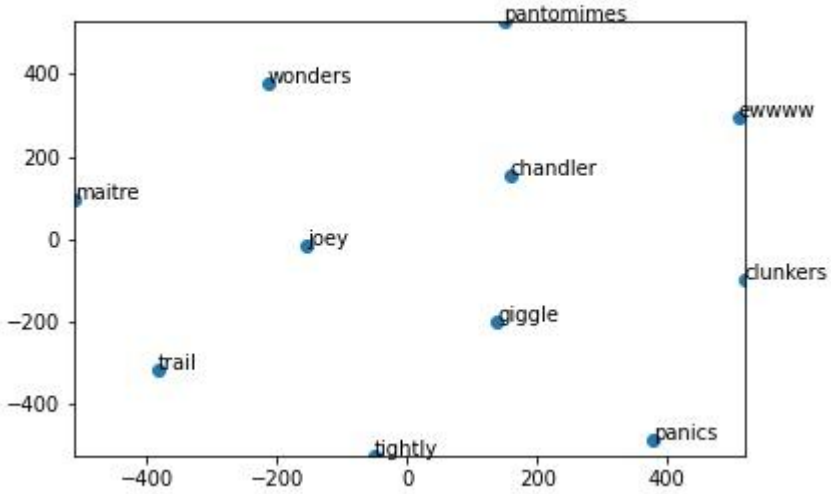


Similar Words to Rachel (50 embeddings):

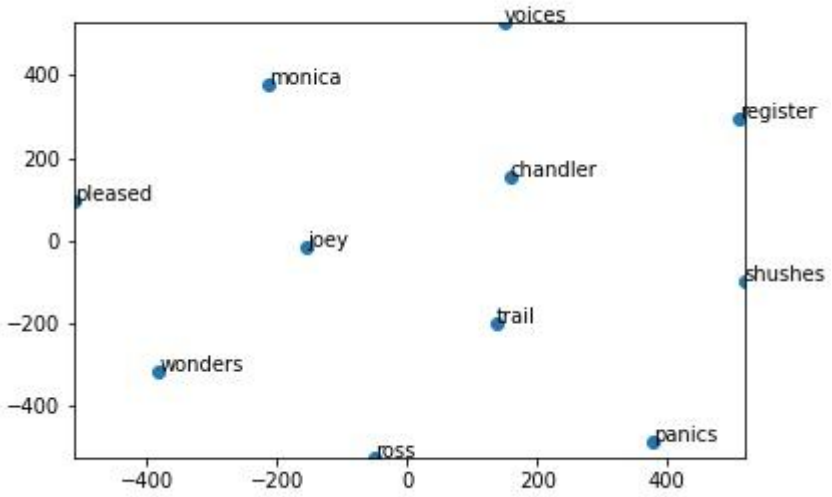


When considering the plots resulting from the three different embeddings, most of the worst change their position. Words as “ross”, “hence”, “laren” and “panics” are present in all three diagramas, while other words only appear in some of them. For example, “ohio”, where Rachel was from, only appears in the plot generated with the embeddings with 50 features. Also, Emily, the woman with whom Ross was engaged, only appears in the last diagram. The other characters, such as Chandler and Phoebe aslo only appear in this case. In fact, we could identify a bigger similarity from the plotted words in this last diagram.

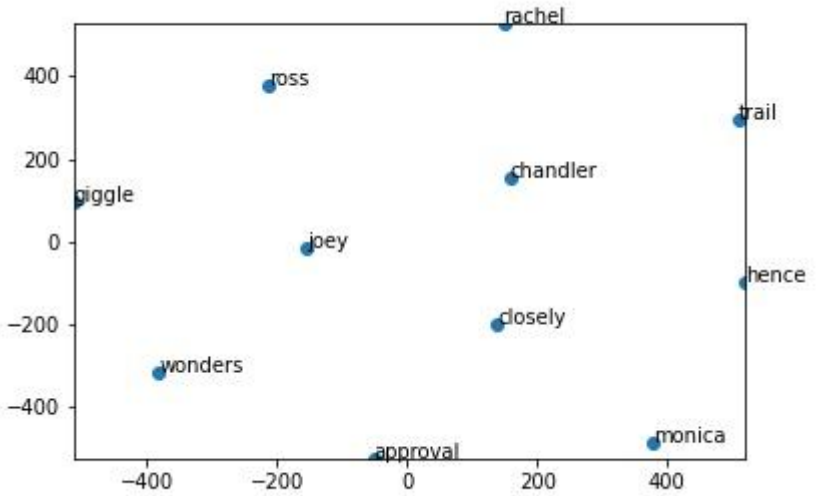
Similar Words to Chandler (150 embeddings):



Similar Words to Chandler (100 embeddings):



Similar words to Chandler (50 embeddings)



In the case of Chandler, we can also find more similar words in the case of the plot generated with the embeddings with 50 features. In this case the other main characters such as Ross, Teacher, Joey and Monica appear as some of the most similar words. Also the word “giggle”, that could be used to describe Chandler, appears frequently in all three diagrams.

3. Architectures with pre-trained embeddings

In order to build a classifier to identify the most likely character for an input line of dialogue, we prepared the datasets, created the training, validation, and testing sets, defined 3 architectures for the neural networks that used the previously trained embeddings and built the models.

In order to prepare the dataset, initially we considered only the dialogues that were made by the main characters, meaning Chandler, Joey, Monica, Phoebe, Rachel and Ross. The values were turned into OneHotVectors using the library OneHotEncoder in order to adapt the dataset to the expected format for the multiclass classification task. Its imported to note that the vocabulary

On the other hand, the dataset was divided in three sets, training, testing and validation. This division was done using the train_test_split method by sklearn that split arrays into random train and test subsets. The proportions used for each one of them were 60% for test, 20% for validation and 20% for testing. The number of samples for each subset were:

Subset	Number of Samples
Training	9303
Testing	9303
Validation	27908
Total	46514

Since the goal was to use the pretrained embeddings, it was necessary to compare the vocabulary that was used by the model, that would be defined by the method TextVectorization by TensorFlow, and the vocabulary that the embeddings used. The vocabulary generated by the TextVectorization method contained 14020 words, while there were a total of 8166 embeddings. The method get_weight_matrix was in charge of generating the corresponding translation between the embeddings and the model's vocabulary by creating a matrix where each i-th column corresponded to the embedding of the i-th word of the vocabulary. After comparing both vocabularies, there were a total of 7452 words common in both representations, and 6568 word embeddings missing for words of the model's vocabulary. These missing words were represented with vectors of 0, adding a considerable percentage of error to the models. It's important to note that the dimension of the resulting weight matriz would change according to the particular size of the embedding. The 3 matrix generated hence had the following dimensions:

- 50x14020
- 100x14020
- 150x14020

After building the corresponding weight matrix in order to use the pretrained embeddings, the three architectures defined were all FeedForward networks that had the following layers in common in the beginning:

- Input layer that received int values that corresponds to the index of the word in the vocabulary
- The embedding layer that translated that index to its corresponding embedding

- A GlobalAveragePooling1D layer that generates one feature map for each corresponding category of the classification task in the last layer instead of a dense layer

And in the end:

- A dense layer with one cell for each main character, hence 6, that used the softmax method in order to calculate the particular exit.

On the other hand, each architecture implemented different hidden layers in order to differentiate them.

First Architecture:

- A dense activation layer that had 10 cells and used the ‘relu’ algorithm as the activation function

The summary of the generated models are:

```
red_50_1.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding (Embedding)        (None, None, 50)          701000
_________________________________________________________________
global_average_pooling1d (Gl (None, 50)                0
_________________________________________________________________
dense (Dense)                (None, 10)                510
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 66
=================================================================
Total params: 701,576
Trainable params: 576
Non-trainable params: 701,000
_________________________________________________________________
```

```
red_100_1.summary()
```

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_1 (Embedding)      (None, None, 100)         1402000
_________________________________________________________________
global_average_pooling1d_1 ( (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
_________________________________________________________________
dense_3 (Dense)              (None, 6)                 66
=================================================================
Total params: 1,403,076
Trainable params: 1,076
Non-trainable params: 1,402,000
_________________________________________________________________
```

```
red_150_1.summary()
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_2 (Embedding)      (None, None, 150)         2103000
_________________________________________________________________
global_average_pooling1d_2 ( (None, 150)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1510
_________________________________________________________________
dense_5 (Dense)              (None, 6)                 66
=================================================================
Total params: 2,104,576
Trainable params: 1,576
Non-trainable params: 2,103,000
_________________________________________________________________
```

Considering the given architecture we know that the first layer corresponds to the input layer that receives the word's index. The second layer is of dimension size_vocab x embedding_dim, where size_vocab is the size of the vocabulary (14020) and the dimensions of the vector change according to the size of the embedding (50, 100 or 150). The third layer is just a flat layer of 1 x embedding_dim since it's an average pooling layer. The fourth layer is of size embedding_dim x 10, since there were 6 cells defined and it is a dense layer. However, since we have some bias, the total amount of parameter is given by (embedding_dim+1)*10. The last layer is 10x6, given that there are 6 characters that we want to predict 6 different classes. The parameters in this case are also given by adding the bias to the number of cells of the previous layer.

Second Architecture:

- A dense activation layer that had 80 cells and used the ‘relu’ algorithm as the activation function
- A dense activation layer that had 250 cells and used the ‘relu’ algorithm as the activation function

The summary of the generated models are:

```
red_50_2.summary()
```

```
Model: "model_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_3 (Embedding)      (None, None, 50)          701000
_________________________________________________________________
global_average_pooling1d_3 ( (None, 50)                0
_________________________________________________________________
dense_6 (Dense)              (None, 80)                4080
_________________________________________________________________
dense_7 (Dense)              (None, 250)               20250
_________________________________________________________________
dense_8 (Dense)              (None, 6)                 1506
=================================================================
Total params: 726,836
Trainable params: 25,836
Non-trainable params: 701,000
_________________________________________________________________
```

```
red_100_2.summary()
```

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_4 (Embedding)      (None, None, 100)         1402000
_________________________________________________________________
global_average_pooling1d_4 ( (None, 100)               0
_________________________________________________________________
dense_9 (Dense)              (None, 80)                8080
_________________________________________________________________
dense_10 (Dense)             (None, 250)               20250
_________________________________________________________________
dense_11 (Dense)             (None, 6)                 1506
=================================================================
Total params: 1,431,836
Trainable params: 29,836
Non-trainable params: 1,402,000
_________________________________________________________________
```

```
red_150_2.summary()
```

```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_5 (Embedding)      (None, None, 150)         2103000
_________________________________________________________________
global_average_pooling1d_5 ( (None, 150)               0
_________________________________________________________________
dense_12 (Dense)             (None, 80)                12080
_________________________________________________________________
dense_13 (Dense)             (None, 250)               20250
_________________________________________________________________
dense_14 (Dense)             (None, 6)                 1506
=================================================================
Total params: 2,136,836
Trainable params: 33,836
Non-trainable params: 2,103,000
_________________________________________________________________
```

Considering the given architecture we know that the first layer corresponds to the input layer that receives the word's index. The second layer is of dimension size_vocab x embedding_dim, where size_vocab is the size of the vocabulary (14020) and the dimensions of the vector change according to the size of the embedding (50, 100 or 150). The third layer is just a flat layer of 1 x embedding_dim since it's an average pooling layer. The fourth layer is of size embedding_dim x 80, since there were 80 cells defined and it is a dense layer.

However, since we have some bias, the total amount of parameter is given by (embedding_dim+1)*80. The fifth layer is of size embedding_dim x 250, since there were 250 cells defined and it is a dense layer. However, since we have some bias, the total amount of parameter is given by (80+1)*250. The last layer is 10x6, given that there are 6 characters that we want to predict 6 different classes. The parameters in this case are also given by adding the bias to the number of cells of the previous layer.

Third Architecture:

- A dense activation layer that had 80 cells and used the ‘relu’ algorithm as the activation function
- A dropout layer that had a 30% rate
- A batch normalization layer that applied a transformation that maintained the mean output close to 0 and the output standard deviation close to 1.
- A dense activation layer that had 250 cells and used the ‘relu’ algorithm as the activation function
- A dropout layer that had a 30% rate

The dimensions of each layer are:

```
red_50_3.summary()
```

```
Model: "model_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_6 (Embedding)      (None, None, 50)          701000
_________________________________________________________________
global_average_pooling1d_6 ( (None, 50)                0
_________________________________________________________________
dense_15 (Dense)             (None, 80)                4080
_________________________________________________________________
dropout (Dropout)            (None, 80)                0
_________________________________________________________________
batch_normalization (BatchNo (None, 80)                320
_________________________________________________________________
dense_16 (Dense)             (None, 250)               20250
_________________________________________________________________
dropout_1 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_17 (Dense)             (None, 6)                 1506
=================================================================
Total params: 727,156
Trainable params: 25,996
Non-trainable params: 701,160
_________________________________________________________________
```

```
red_100_3.summary()
```

```
Model: "model_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_8 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_7 (Embedding)      (None, None, 100)         1402000
_________________________________________________________________
global_average_pooling1d_7 ( (None, 100)               0
_________________________________________________________________
dense_18 (Dense)             (None, 80)                8080
_________________________________________________________________
dropout_2 (Dropout)          (None, 80)                0
_________________________________________________________________
batch_normalization_1 (Batch (None, 80)                320
_________________________________________________________________
dense_19 (Dense)             (None, 250)               20250
_________________________________________________________________
dropout_3 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_20 (Dense)             (None, 6)                 1506
=================================================================
Total params: 1,432,156
Trainable params: 29,996
Non-trainable params: 1,402,160
_________________________________________________________________
```

```
red_150_3.summary()
```

```
Model: "model_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         [(None, None)]            0
_________________________________________________________________
embedding_8 (Embedding)      (None, None, 150)         2103000
_________________________________________________________________
global_average_pooling1d_8 ( (None, 150)               0
_________________________________________________________________
dense_21 (Dense)             (None, 80)                12080
_________________________________________________________________
dropout_4 (Dropout)          (None, 80)                0
_________________________________________________________________
batch_normalization_2 (Batch (None, 80)                320
_________________________________________________________________
dense_22 (Dense)             (None, 250)               20250
_________________________________________________________________
dropout_5 (Dropout)          (None, 250)               0
_________________________________________________________________
dense_23 (Dense)             (None, 6)                 1506
=================================================================
Total params: 2,137,156
Trainable params: 33,996
Non-trainable params: 2,103,160
_________________________________________________________________
```

Considering the given architecture we know that the first layer corresponds to the input layer that receives the word's index. The second layer is of dimension size_vocab x embedding_dim, where size_vocab is the size of the vocabulary (14020) and the dimensions of the vector change according to the size of the embedding (50, 100 or 150). The third layer is just a flat layer of 1 x embedding_dim since it's an average pooling layer. The fourth layer

is of size embedding_dim x 80, since there were 80 cells defined and it is a dense layer. However, since we have some bias, the total amount of parameter is given by (embedding_dim+1)*80. The fifth layer is also a flat layer, since it's a Dropout Layer, hence 80x1. The sixth layer is of dimensions 80x4 given that it is a batch normalization layer. The seventh layer is a dense layer, with 80x250 dimensions, and the parameters are calculated taking into account the bias (80+1)x250. The eight layer is also a flat layer, since it's a Dropout Layer, hence 250x1. The last layer is 250x6, given that there are 6 characters that we want to predict 6 different classes. The parameters in this case are also given by adding the bias to the number of cells of the previous layer.

When comparing the results of combining the 3 architectures with the 3 types of embeddings in terms of accuracy, precision and recall in training, validation, and testing sets, the following table is useful considering that the precision and recall are only for the label 5, since it's the only class that was predicted among the labels, and the accuracy in regard to the model:

Model	Accuracy	Precision	Recall
Architecture 1 - 50	17%	17%	100%
Architecture 1 - 100	17%	17%	100%
Architecture 1 - 150	17%	17%	100%
Architecture 2 - 50	17%	17%	100%
Architecture 2 - 100	18%	18%	100%
Architecture 2 - 150	18%	18%	100%
Architecture 3 - 50	18%	18%	100%
Architecture 3 - 100	18%	18%	100%
Architecture 3 - 150	18%	18%	100%

These values were calculated using the sklearn metrics confusion matrix and classification report. Just as keras programs the models, these calculations were done using batch processing. We can see that the models' recall is pretty high, while the accuracy and precision are pretty low. However it's important to note that the values are all pretty similar for all the architectures and for all the types of embeddings.

4. Architectures with training layer

5. Comparison