Project Report:
Make Your Own Neural Network for NLP

Jenia Kim

February 2020

1 Introduction

The book 'Make Your Own Neural Network' (Rashid, 2016) takes the reader on a step-by-step journey to build a simple neural network from scratch. The author offers accessible explanations of the mathematical concepts involved and then implements them in Python, using the famous MNIST dataset of handwritten numbers as an example classification task.

In this project, I followed the book and created my own neural network, while using an NLP task – classification of movie plots by genre – instead of image recognition. This report describes the steps I've taken and the results of my experiments. The goals of the exercise were (a) to understand how a simple feed-forward neural network works by implementing it from scratch, and (b) to familiarize myself with a new (for me) way of text representation for machine learning: `doc2vec` (Le and Mikolov, 2014).

The code for this project is available at https://github.com/vanboefer/nn_doc2vec_exercise.

2 Data

I use a small dataset of movie plots labeled with their genre.[1] Table 1 shows the class distribution and the mean text lengths (excluding punctuation and stopwords) in the dataset. As evident, the dataset is not balanced; 32% of the records belong to the largest class ('comedy'), while only 8% of the records belong to the smallest class ('fantasy'). In terms of mean text length, there are no significant differences between the classes.

[1] https://github.com/RaRe-Technologies/movie-plots-by-genre/tree/master/data

Genre	No. docs	%	Mean length
comedy	780	32%	37.1
action	437	18%	38.5
romance	380	16%	42.7
sci-fi	352	15%	39.0
animation	283	12%	37.0
fantasy	195	8%	42.1
TOTAL	2,427	100%	38.9

Table 1: Class distribution & text lengths in the movies dataset

animation		fantasy		comedy		action		romance		sci-fi	
token	*count*	*token*	*count*	*token*	*count*	*token*	*count*	*token*	*count*	*token*	*count*
world	68	world	53	life	177	one	96	love	160	earth	112
young	63	one	41	one	160	man	79	life	138	world	86
new	51	must	41	new	153	new	73	one	94	one	84
named	45	young	40	man	125	life	68	young	87	alien	77
one	45	life	40	two	121	must	66	new	74	planet	76
must	42	evil	38	get	119	find	63	woman	71	new	71
get	39	king	34	find	102	agent	58	man	69	future	68
two	39	father	34	family	94	world	54	two	67	find	66
life	37	family	31	father	94	young	53	time	56	human	52
boy	36	find	30	young	88	team	50	finds	55	life	49

Table 2: Top 10 tokens by genre

Table 2 shows the 10 most frequent tokens (excluding punctuation and stopwords) per genre. Some tokens – including *life*, *young*, *new* and *world* – are frequent across genres. Other tokens are frequent in only one specific genre, e.g. *boy* ('animation'), *king* ('fantasy'), *agent* ('action'), *love* ('romance') and *alien* ('sci-fi').

The raw text of the movie plots was converted to lowercase and tokenized with NLTK[2]; punctuation and stopwords were removed. The resulting lists of tokens were used in the following steps.

[2] https://www.nltk.org/

3 Doc2Vec

3.1 Introduction

Machine learning algorithms, including neural networks, require the input (in our case, the text of the movie plots) to be represented in a fixed-length vector. One common representation is the 'bag-of-words': a vector where each dimension represents a word in the vocabulary. If a word is present in a document, its dimension in the vector is marked with 1 or with its (weighted) frequency; otherwise, its dimension is marked with 0. This representation has three main disadvantages: (a) the order of the words in a document is not represented, (b) the meaning of the words is not represented, i.e. each word is equally similar to any other word in the vocabulary, and (c) high dimensional sparse vectors are less suitable inputs for neural networks, which work better with dense inputs.

A popular solution to the two latter issues is creating dense word vectors (or 'word embeddings') that represent the distributional semantics of words. This can be done, for example, by training a model in which a context (e.g. three consecutive words) is used to predict the next word. As an indirect result of this prediction task, representations of words are created that capture certain aspects of meaning, like similarity between words (since similar words tend to appear in similar contexts).[3] One famous model that creates such word embeddings is `word2vec` (Mikolov et al., 2013).

To represent documents of varying lengths using word embeddings, one can, for example, use a weighted average of all the word vectors in the document. This approach, however, still suffers from the weakness of ignoring the word order, just like the bag-of-words. As a solution, Le and Mikolov (2014) propose the 'Paragraph Vector' or `doc2vec` approach, which is very similar to the `word2vec` method. In this framework, there are two possible training methods (i.e. two possible prediction tasks): the Distributed Memory Model of Paragraph Vectors (PV-DM) and the Distributed Bag of Words Model of Paragraph Vector (PV-DBOW).

In PV-DM, the task is to predict the next word, given a context. The only difference in comparison to `word2vec` is that in addition to the context words, another 'word' is introduced which represents the document (i.e. a document id). The vectors of the context words and the vector of the document are averaged or concatenated to predict the next word. The fixed-length contexts are sampled from a sliding window over the document; the

[3] By "indirect result", I mean that we do not use the actual outputs of the prediction task; rather, we take the learned classifier weights and use them as word embeddings.

document vector is shared across all contexts generated from one document, but not across documents. The word vectors, on the other hand, are shared across documents, i.e. the vector for *king* is the same for all documents.

In PV-DBOW (analogous to `word2vec`'s Skip-Gram), the task is to distinguish between a real sequence of words sampled from the document (e.g. *two best friends seek*) and a sequence of randomly sampled words (e.g. *girl-friends local friends best*). The input is the document vector only. This model is faster to train, since it doesn't need to store individual word vectors.

In both training methods, the weights learned in the classification process are used as the representation of the document as a whole. These representations inherit an important property of word embeddings: they capture the distributional semantics of the words. In addition, they take into account the word order, at least in the small contexts of the sampling window.

3.2 Training and evaluating `doc2vec` models

3.2.1 Evaluation through performance on the main task

In this section, I experiment with two different settings for training a `doc2vec` model: vector size and training method (PV-DM vs. PV-DBOW). I use the entire movies dataset to create the `doc2vec` representations and then apply these representations in the main task: classification of the documents by genre.[4] The classification accuracy is used to evaluate the `doc2vec` models.

The results are summarized in Table 3; there are 10 models combining five different vector sizes with the two possible training methods. For example, in the first row we see the performance of a 400-dimension vector trained with PV-DM and a 400-dimension vector trained with PV-DBOW.

The first three columns provide the settings for the `doc2vec` training; I use the default settings of the `gensim` implementation[5], except for the ones indicated in the table: I train for 40 epochs and use only words with minimum frequency of 2. The four columns under 'neural network' specify the settings of the neural network classifier for the main task; they are kept constant, since the experiment focuses on the `doc2vec` representations. The number of the nodes in the hidden layer is set to 2/3 of the number of the

[4] Since the classes (i.e. the movie genres) are not used in any way during the `doc2vec` training, I use the whole dataset for creating the model and split to train / test only for the main task.

[5] https://radimrehurek.com/gensim/models/doc2vec.html#gensim.models.doc2vec.Doc2Vec

doc2vec			neural network				accuracy	
epochs	min_count	**vec_size**	hid_nodes	learn_r	epochs	reinfer	**PV-DM**	**PV-DBOW**
40	2	**400**	270	0.1	40	False	**0.494**	**0.486**
40	2	**300**	200	0.1	40	False	**0.469**	**0.506**
40	2	**200**	130	0.1	40	False	**0.457**	**0.486**
40	2	**100**	70	0.1	40	False	**0.486**	**0.490**
40	2	**50**	30	0.1	40	False	**0.506**	**0.523**

Table 3: doc2vec: Experiments with vector size and training method

input nodes (the vector size), the learning rate is set to 0.1, the number of training epochs is set to 40, and the 'reinfer' parameter is set to 'False', meaning that document vectors stored in the doc2vec model are used 'as is' and not re-inferred during classification. For further explanations and experimentation with the neural network settings, see Section 4.

The results in Table 3 show that the PV-DBOW training method tends to perform better than the PV-DM method on this data, for all vector sizes. Further, the best-performing vector size for both methods is of 50 dimensions. Both findings could be related to the fact that the documents in the dataset are short, but it is hard to pinpoint the reason for sure.

Since the results of the main classification task vary from run to run (due to random initialization), it makes sense to verify that the conclusion is stable. Therefore, I did five runs of classification with the 50-dimension models: the mean accuracy of the runs with the PV-DBOW_50 model was 0.521 (std: 0.011), while the mean accuracy of the runs with the PV-DM_50 model was 0.491 (std: 0.012). Based on this, I chose the PV-DBOW_50 model as the preferred document representation; this is the model used in all the following sections.

3.2.2 Evaluation through self-similarity

Another way to assess the quality of a doc2vec model is by re-inferring new vectors for all the documents in the dataset and comparing the inferred vectors with the original vectors stored in the model.[6] If the model is good, we expect the inferred vector of a document to be most similar to itself, i.e. to the original vector of the same document.

[6] https://radimrehurek.com/gensim/auto_examples/tutorials/run_doc2vec_lee.html#sphx-glr-auto-examples-tutorials-run-doc2vec-lee-py

Performing this evaluation on the chosen PV-DBOW_50 model, the results are very good: out of the 2,427 documents in the dataset, 2,413 (99.4%) are most similar to themselves. Only 11 documents (0.5%) are more similar to another document and rank themselves as the second most similar; the 3 remaining documents rank themselves in a lower-than-second place. An example of a "mistake" that the model made is shown below; the inferred vector of document (a) is more similar to the original vector of document (b) than to the original vector of itself:

> `(a) A recalcitrant thief vies with a duplicitous Mongol ruler for the hand of a beautiful princess.`
>
> `(b) Prince Ahmad is the rightful King of Bagdad but he has been blinded and cast out as a beggar. Now a captive of the wicked Grand Vizier Jaffar he is cast into a dungeon where he meets Abu, the best thief in all Bagdad. Together they escape and set about a series of adventures that involve a Djinni in a bottle, a mechanical flying horse, an all-seeing magic jewel, a flying carpet and a beautiful princess.`

The source of the mistake is quite transparent: document (a) is very short and contains at least three very rare words (*recalcitrant*, *vies*, *duplicitous*) that are probably not taken into account by the model (as mentioned in Section 3.2.1, the model only considers words that appear at least twice in the dataset); the (sequences of) words that are considered by the model, e.g. *thief*, *beautiful princess*, appear also in text (b). It is also worth noting that the similarity scores of the first two places are quite high: the similarity of (a) to (b) is 93.3% and the similarity of (a) to itself is 91.8%.

To sum up, this evaluation suggests that the chosen PV-DBOW_50 model is a good representation of the documents. In the next section, it is utilized for experimentation with the neural network on the main classification task.

4 Classification experiments

In this section, I experiment with the settings of the neural network in order to see how they affect the performance on the classification task. The original dataset is split into train and test data (0.9/0.1, respectively); the distribution between classes in the train and test sets is proportional to the distribution in the whole dataset (test: *comedy*: 30%, *action*: 23%, *sci-fi*: 15%, *animation*: 12%, *romance*: 10%, *fantasy*: 10%). The task of the classifier is to predict the genre given the `doc2vec` representation of the plot.

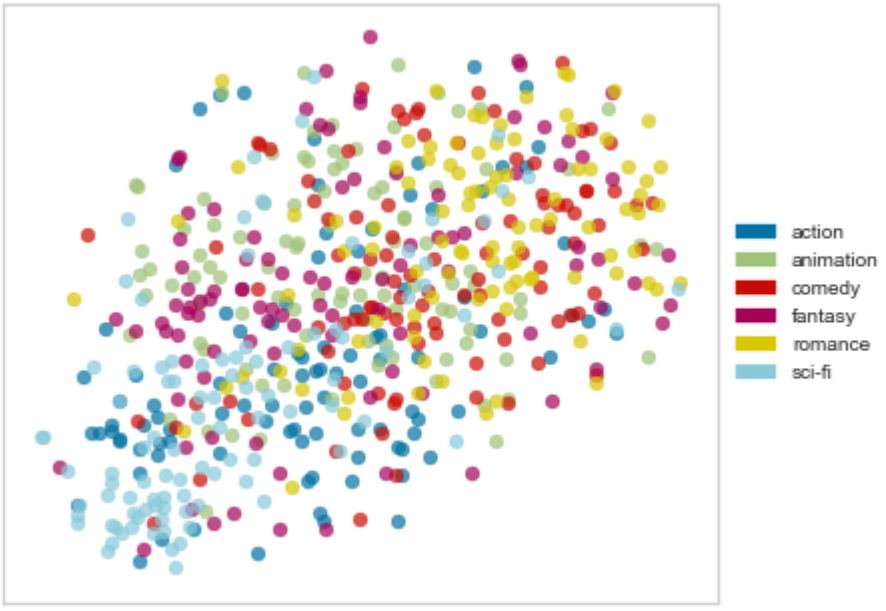

Figure 1: t-SNE plot of a sample of 100 documents from each class

To get an indication of how difficult the classification task is, we can use t-SNE: a non-linear dimensionality reduction technique, used for visualization of high-dimensional data in a two-dimensional space (Maaten and Hinton, 2008). t-SNE models each high-dimensional object in such a way that similar objects appear nearby each other in the two-dimensional space, while dissimilar objects appear far away from each other. In our case, each `doc2vec` vector is a high-dimensional object, so the algorithm groups together document vectors that are similar to each other, while document vectors that are different end up further from each other on the plot.

Figure 1 shows the visualization for randomly sampled 100 documents from each class (the sampling is done to make the visualization clearer; too many data points make the plot too dense to see the colors). We see that document vectors from different classes (different colors) are scattered all over the plot and are mixed with each other; the only evident color clustering is the sci-fi cluster in the left-bottom corner of the plot. This suggests that the classification task is quite difficult, since the document vectors of the same class are not necessarily more similar to each other.

	precision	recall	f1-score	support
action	0.679	0.345	0.458	55
animation	0.600	0.400	0.480	30
comedy	0.495	0.736	0.592	72
fantasy	0.400	0.083	0.138	24
romance	0.478	0.440	0.458	25
sci-fi	0.500	0.811	0.619	37
accuracy			0.523	243
macro avg	0.525	0.469	0.457	243
weighted avg	0.539	0.523	0.493	243

Table 4: Classification report: baseline NN

4.1 Baseline neural network

The baseline neural network is constructed according to the book 'Make Your Own Neural Network' (Rashid, 2016). It is a simple fully connected feed-forward neural network, with one hidden layer. The initial weights are randomly sampled from a normal distribution centered around zero and with a standard deviation $1/\sqrt{(\text{number of incoming links})}$. The activation function is the sigmoid function.

The default settings of the neural network, used for the experiments in Section 3.2.1, are: learning rate of 0.1, number of nodes in the hidden layer is 2/3 of the number of input nodes (i.e. 30 for the 50-dimension input vectors that I use), 40 training epochs, document vectors from the model are used 'as is' (i.e. not re-inferred). The performance of this baseline neural network with the chosen PV-DBOW_50 `doc2vec` model is summarized in Table 4. We see that the network performs best on the 'sci-fi' class (F1-score 0.619), as expected from the t-SNE visualization; it also performs quite well on the biggest class 'comedy' (F1-score 0.592); the poorest performance is on the smallest 'fantasy' class (which constitutes only 8% of the whole dataset). The overall accuracy on the test set is 0.523.

In the following sections, I experiment with the default settings to examine how they affect the performance on the task. The `doc2vec` representations used for all the experiments is the PV-DBOW_50 model.

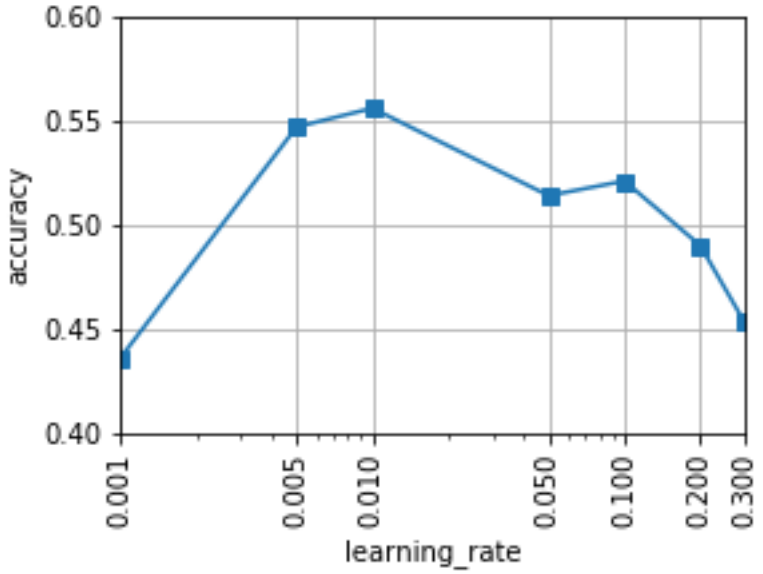

Figure 2: Classification accuracy for 7 learning rates (logarithmic scale)

4.2 Experimenting with the learning rate

Figure 2 shows the results of experimenting with seven different learning rates, while keeping the rest of the settings as in the baseline.[7] The best results are obtained with 0.005 (accuracy: 0.547) and 0.01 (0.556). As mentioned above, the results vary from run to run (due to random initialization); therefore, to decide between the two rates, I did five runs with each rate and averaged the results. The average accuracy for a learning rate of 0.01 is 0.543 (std: 0.0083) and for a learning rate of 0.005 is 0.552 (std: 0.0097). Based on these results, I chose a learning rate of 0.005 as the optimal and used it in the consequent experimentation.

4.3 Experimenting with the number of hidden nodes

The number of nodes in the hidden layer needs to be somewhere between the number of input nodes (50 in our case) and the number of output nodes (6 in our case). Figure 3 shows the results for seven different options that I tried. It seems that any number between 25 and 45 gives good results

[7] In this and the following figures, the plotted results are from one run. I report averaged results from several runs in the text.

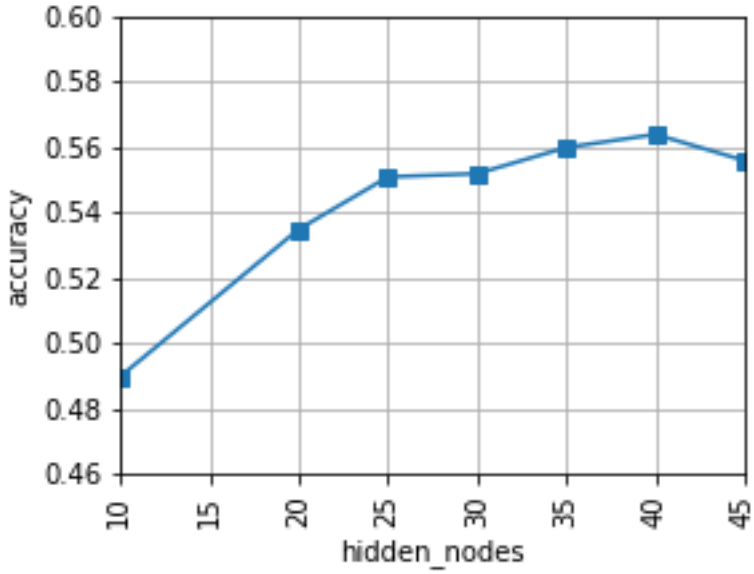

Figure 3: Classification accuracy for different number of hidden nodes

(accuracy of 0.551-0.564). To get a more stable impression I did five runs with 35 and 40 nodes, which resulted in accuracy of 0.551 (std: 0.008) and 0.549 (std: 0.0068), respectively. We already have an average of five runs with 30 nodes from the previous section: 0.552 (std: 0.0097), which turns out to be the best option. In other words, the optimal number of hidden nodes is the one used in the baseline, i.e. 30. This is also compatible with a popular rule of thumb that recommends the number of nodes in the hidden layer to be 2/3 of the number of input nodes.

4.4 Experimenting with the number of epochs

An epoch is one run (forward and backward) through the entire training data that the neural network performs during training. On the one hand, the more epochs the better, since each run helps the gradient descent by providing more chances to reach the minimum. On the other hand, too many epochs might result in overfitting the training data and consequently not generalizing well to unseen data, and they might also take a long time. Therefore, I experiment with a few options to find the optimal number. The results are shown in Figure 4; it seems that 40 or more epochs give good results. The average accuracy for 40 epochs is already known from

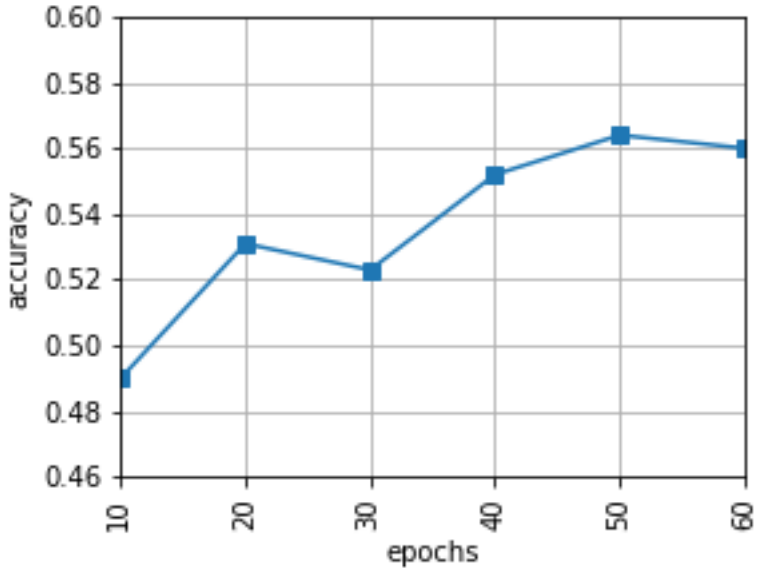

Figure 4: Classification accuracy for different number of training epochs

the previous sections: 0.552 (std: 0.0097); the average over five runs with 50 epochs is 0.553 (std: 0.0097) and with 60 epochs is 0.552 (std: 0.0101). In other words, there is almost no difference between these three options; therefore, I decided to stick with the more time-efficient option of 40 epochs.

4.5 Experimenting with re-inferring the input vectors

During the training of the `doc2vec` model, the representations of the documents are saved as part of the model. These representations can be then used in the classification task. Alternatively, the model can be used to re-infer new vectors for the documents, as was done in Section 3.2.2, and the re-inferred vectors can be used for classification. In this section, I experiment with this setting. Based on the what was shown in Section 3.2.2, i.e. that 99.4% of the re-inferred vectors are most similar to themselves, I did not expect this setting to make a big difference for the results. Indeed, five runs with re-inferred vectors result in an average accuracy 0.552 (std: 0.0057), which is exactly the same as the average of five runs with the original vectors. Since re-inferring the vectors takes time and does not affect the results, this setting is best set to 'False'.

	precision	recall	f1-score	support
action	0.620	0.564	0.590	55
animation	0.800	0.267	0.400	30
comedy	0.496	0.806	0.614	72
fantasy	0.500	0.125	0.200	24
romance	0.455	0.400	0.426	25
sci-fi	0.605	0.622	0.613	37
accuracy			0.547	243
macro avg	0.579	0.464	0.474	243
weighted avg	0.574	0.547	0.522	243

Table 5: Classification report: final NN

4.6 The final neural network

The final neural network has the following settings: learning rate 0.005, 30 hidden nodes, 40 training epochs, `doc2vec` vectors are not re-inferred.

The classification results of one example run with this network are shown in Table 5. Compared to the baseline in Table 4, we can see that the classifier has an overall better performance, mainly due to a big improvement on the 'action' class (from F1-score 0.458 to 0.59), and on the smallest class 'fantasy' (from F1-score 0.138 to 0.2). The best performance of the final neural network is still on the 'sci-fi' class (where the documents are similar to each other and distinct from other classes, see Figure 1) and on the biggest class 'comedy' (for which the classifier has the most examples).

Figure 5 shows the confusion matrix for this run, where we can see the mis-classification mistakes. For the 'comedy' class, the precision is quite low (0.496) meaning that many documents that the classifier labels as 'comedy' actually belong to other classes; the matrix shows us that the main confusion is with the 'action' class, which is not surprising since it's the second biggest class in the data. Another class that has an especially low precision is 'romance' (0.455); this genre tends to be mis-classified as 'comedy'.

5 Conclusion

In this project, I experimented with creating `doc2vec` representations of movie plots and using these representations for a classification task into six genres. The classifier was a simple feed-forward neural network that I implemented from scratch based on the book 'Make Your Own Neural

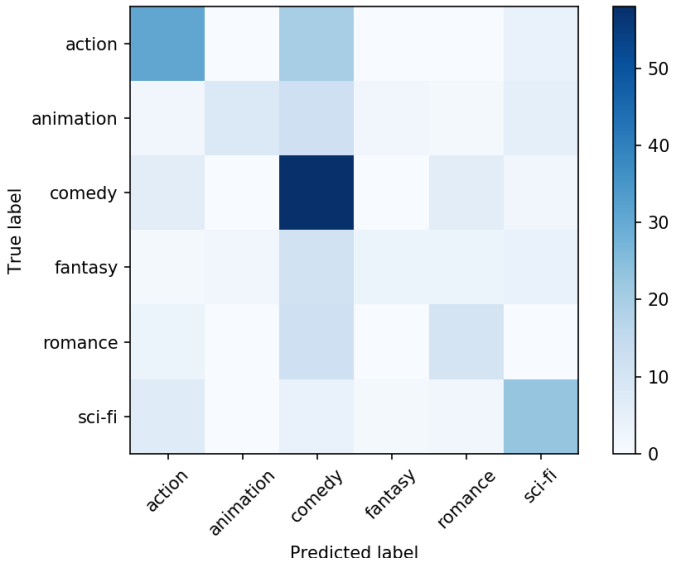

Figure 5: Confusion matrix: final NN

Network' (Rashid, 2016). Although the results of the classification are not great, which is to be expected with such a simple network and such a small dataset, the exercise was very valuable in terms of learning about `doc2vec` and understanding in-depth the principles on which neural networks are based.

References

Le, Q. and Mikolov, T. (2014). Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196.

Maaten, L. v. d. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Rashid, T. (2016). *Make your own neural network*. CreateSpace Independent Publishing Platform.