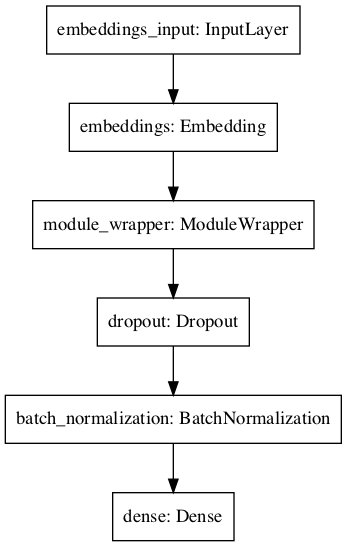
**Word Embeddings**

A word embedding is a learned representation for text where words that have the same meaning have a similar representation. Each word is represented by a real-valued vector, often tens or hundreds of dimensions. Hence, lower dimensionality vocabulary size and computational issues can be lowered. Hence the uniqueness of our dataset we used an own-trained approach for our embeddings. This can outperform pre-trained models, when there is just enough data [Qui et al., 2018]. The embeddings where created by word2vec, fasttext, doc2vec from gensim and an embedding layer from keras and tensorflow. Training the embeddings was based on the whole dataset (19244 reviews). We kept the 1000th most common features. Analyzing the classifiers was done on Random Forest, Logistic Regression, GausianNB, BernoulliNB, LinearSVC, SVC and the RNN (all by default). The embeddings created by the embedding layer were only used by the RNN.

**Recurrent Neural Network**

We use the **RNN** architecture, hence sequenced text data. The **Sequential API** is used. Because it's most suitable for our problem. We neither have multiple inputs from different sources (only text data) nor have we multiple outputs (binary labels). The first layer of our model is the **Embedding Layer**. There the features are generated. The Embedding layer has a size of input\_dim x output\_dim. Input\_dim is the number of most instinct and common words in our case the corpus has 299647 words after preprocessing; we keep (arbitrarily) the 1000 most common unique words (no studies on most instinct and common words for our problem case and in general were found). For the output\_dim the most common rule of thumb is 300 (e.g. Mikolow et al., 2013). However, we’re setting size to 256 (2 to the power of 8). Power of 2 will increase cache utilization during data movement, thus reducing bottlenecks. We want to point out, that Patel & Bhattacharyya (2017) came up with a mathematical approach to calculate the best output\_dim size for word embeddings ('lower bound'). The input length is equal to the MAX\_LENGTH (in our case 260) of the sequence in order to avoid loss of information. Next, we add the **Long-Short-Term-Memory Layer (LSTM)** Layer. This layer is very suitable for semantic parsing (Jia, Robin; Liang, Percy (2016). "Data Recombination for Neural Semantic Parsing". arXiv:1606.03622). The rule of thumb for calculating the hidden nodes (number of neurons in the LSTM) is hidden\_nodes := 2/3 \* (timesteps \* input\_dim). In our case 9013 hidden nodes would be needed (2/3 \* 260 \* 52). Because of computational reasons we only use 500 hidden nodes. Timesteps is the review with the most words (= max of seq\_lengths.describe() see code) which in our case is 260. For the hyperparameter input\_dim, we choose 52. This equals all the lower and upper chars in the alphabet. Moreover, every LSTM layer should be accompanied by a Dropout Layer. This layer will help to prevent overfitting by ignoring randomly selected neurons during training, and hence reduces the sensitivity to the specific weights of individual neurons. 20% is often used as a good compromise between retaining model accuracy and preventing overfitting. Every LSTM layer should be accompanied by a **Dropout layer**. Dropout is a regularization technique for neural network models, against specialization proposed by Srivastava, et al., 2014. This layer will help to prevent overfitting by ignoring randomly selected neurons during training, and hence reduces the sensitivity to the specific weights of individual neurons. 20% is often used as a good compromise between retaining model accuracy and preventing overfitting. Next, a **BatchNormalization Layer** is added. Although there is a debate whether normalization is necessary, the authors of BatchNormalization Layer say, that it should be applied immediately before the non-linearity of the current layer (Hyv¨arinen & Oja, 2000). They say, that "it is likely to produce activations with a stable distribution.". Also Prof. Andrew Ng prefers to add the layer before nonlinearity (activation). Lastly. we add the **Dense Layer**. For For the model, we used the binary cross-entropy and Adam optimizer, as well as the sigmoid activation function, hence the binary classification problem type. The batch size was set to 32. Across a wide range of experiments the best results have been obtained with this batch size (Revisiting Small Batch Training for Deep Neural Networks, 2018).



**Analyses of Classifiers:**

**Word2Vec**

Most unique features were i.e. “work”, “people”, “job”. Least unique features were i.e. “pound”, “disposal”, “Tupperware”.

|  |  |
| --- | --- |
| **Classifier** | **F1-Score** |
| Random Forest | 0.71 |
| Logistic Regression | **0.72** |
| GaussianNB | 0.65 |
| BernoulliNB | 0.70 |
| LinearSVC | 0.71 |
| SVC | 0.71 |

**FastText**

Most frequent features were i.e. “work”, “great”, “good”. Least unique features where i.e. “underpay”, “advantages”, “shortcomings”.

|  |  |
| --- | --- |
| **Classifier** | **F1-Score** |
| Random Forest | 0.71 |
| Logistic Regression | **0.74** |
| GaussianNB | 0.66 |
| BernoulliNB | 0.71 |
| LinearSVC | 0.59 |
| SVC | 0.59 |

**Doc2Vec**

Most frequent features were i.e. “work”, “great”, “good”. Least unique features where i.e. “sunrise”, “beverage”, “scanning”.

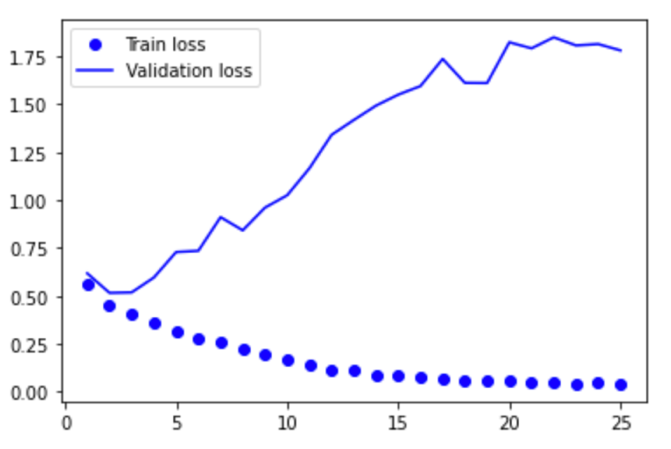
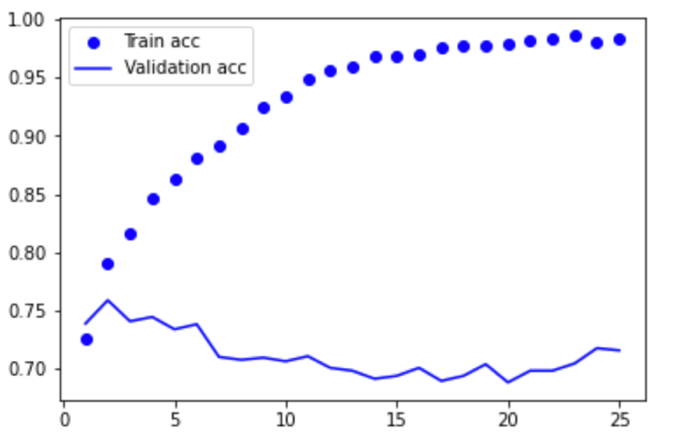
|  |  |
| --- | --- |
| **Classifier** | **F1-Score** |
| Random Forest | 0.72 |
| Logistic Regression | **0.73** |
| GaussianNB | 0.62 |
| BernoulliNB | 0.70 |
| LinearSVC | 0.68 |
| SVC | 0.68 |

Vectorizing with TFIDF increased every F1-Score of every classifier

|  |  |
| --- | --- |
| **Classifier** | **F1-Score** |
| Random Forest | 0.78 |
| Logistic Regression | **0.80** |
| GaussianNB | 0.62 |
| BernoulliNB | 0.76 |
| LinearSVC | 0.78 |
| SVC | 0.78 |

Since Logistic Regression turned out to be the best Classifier we ran a 5 fold grid search with penalty := l1, l2, elasticnet, none; C := -4, 4, 20; solver := liblinear, sag and saga with the whole dataset. Analyzing with the pruned hyperparameters (C := 4, solver := sage, penalty = l1) did not increased, but decreased F1 (0.78).

**RNN**

Increasing training accuracy but decreasing validation accuracy. The larger the gap, the higher is overfitting and specialization. The training loss keeps decreasing after every epoch. Our model is learning to recognize the specific reviews in the training set. The validation loss keeps decreasing at first, but increases after every epoch. Our model is not generalizing well enough on the validation set after one epoch; however, a very slight drop is seen right at epoch 25. More epochs and data would be needed, to clarify this trend. A good fit is identified by a training and validation loss that decreases to a point of stability with a minimal gap between the two final loss values. This occurs after 1 epoch (loss := 79,79%). There the accuracy is decreasing after 1 epoch (f1 := 72,54%). One epoch should be used for initializing our model, hence avoiding specialization. The model was trained with 10000 reviews and 301355 words.