TDT4171 Artificial Intelligence Methods
Assigment 5

Odd André Owren

March 2020

Feature extraction

```
vectorizer = HashingVectorizer(stop_words="english",binary=True)
```

Listing 1: Implementation of Feature Extraction

Naive Bayes and Decision Tree both have in common that they are dependent on a feature extraction to give a representation of presence or absence of certain words in the text. This does not say anything about the context and where in a sentence a word is, just whether or not it is present. This was done with a package for feature extraction in text from sklearn. With the HashingVectorizer, most of the parameters set as default are good, but I chose to use two parameters to increase the effectiveness of this feature extraction.

The first parameter is the stop-words paramater:

> If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if `analyzer == 'word'`. [1]

This parameter was set to `"english"`, as the reviews in the dataset were in English.

The other parameter was the binary parameter. This is a parameter that sets all non-zero values to 1. "This is useful for discrete probabilistic models that model binary events rather than integer counts." [1].

Naive Bayes

```
BernoulliNB(alpha=0.3)
```

Listing 2: Implementation of Naive Bayes

In the implementation of a Naive Bayes classifier for a multivariate Bernoulli model, the main parameter to focus on is the smoothing parameter, α. This is

an additive Laplace smoothing parameter. When setting α different to 0, we get that no probability is ever 0. By default, this value is set to 1. The closer α is to 0, the higher the bias towards words that are already recognized by the model. This means that it will further reinforce already known words that reoccur multiple times, rather than extensively extend it's own vocabulary.

In the implementation this was chosen as 0.3 after some performance testing around different values of this α. The reason that the highest precision is around this value is as described above, that there will be a bias towards often used words. Since the English language is limited (in the sense of roughly 400 000 reviews), most positive or negative words will be used multiple times, and thus will over time become rather safe indicators whether or not a review is positive.

The final accuracy of the Naive Bayes model was at **0.849626**, or **84.96%**.

Decision Tree

```
tree.DecisionTreeClassifier(max_depth=10)
```

Listing 3: Implementation of Decision Tree

In the implementation of the Decision Tree, there were one parameter chosen, `max_depth`. This is purely chosen to limit the computation power required. Without a limit here and letting the decision tree expand all nodes until they are pure, the training of the model takes a lot of time. `max_depth` was set to 10. All the other default parameters that are set when implementing the model gave satisfactory results.

The final accuracy of the Decision Tree was **0.8094430** or **80.94%**.

Recurrent Neural Network

```
model = Sequential()

model.add(Embedding(input_dim=vocab_size, output_dim=64,
    input_length=max_length))
model.add(LSTM(64))
model.add(Dense(2, activation='sigmoid'))

model.compile(optimizer=RMSprop(), loss='binary_crossentropy',
    metrics=['accuracy'])
```

Listing 4: Implementation of Recurrent Neural Network

When choosing the parameters in this part, I had to make some sacrifices and trade-offs between performance and resource-management, more specifically time.

Firstly, with the variable `max_length`, there were possibilities to experiment with this to gain performance. Because of the huge dataset, the estimated time

to train the model with a full length dataset was around 6 hours per epoch. By cutting the max length of each review to $\frac{1}{20}$ of the original length, I seem to have gotten a good trade-off between length and information, and the training time per epoch went down from 6 hours to 2 minutes.

Secondly, when building the recurrent neural network it was made up of 3 different layers: embedding, LSTM and dense. The embedding had one node per entry of the vocabulary, which in total had 1062. Furthermore, I had to set the output to 64 to limit the resource use. This is the same as the input of LSTM, the layer that gets its input from the embedding. Here the output was 2, which again gives the final layer, the dense nodes, an input of 2. By using two nodes in the final layer one is able to get a vector on the form of `[positive,negative]`, which is better than a pure binary result. This also presupposes that the `y_data` is of this form. To do this, I used `to_categorical` from the keras.utils-package. This takes the 1-dimensional results from the dataset and turns it into a 2-dimensional result vector to compare with the result from the network.

Finally, when compiling the model, I used the `RMSprop()`-optimizer, which is considered a great optimizer for recurrent neural networks. For the loss-function, I went with `binary_crossentropy`, as this is a good function when using binary result vectors, i.e. two classes (in this case positive and negative).

The resulting accuracy and loss from the model was:
Accuracy: **0.923047**
Loss: **0.186883**
I view this result as satisfactory, as one should expect at least a 90% accuracy with this model.

Reasons for improvement between models

The improvement for the recurrent neural network comes mainly from the fact that one is able to represent context through the neural network. Expressions such as "like" and "not like" can be distinguished, as well as possible double negatives throughout a sentence. The input form of a node for each entry in the vocabulary also drastically improves the means of representation through the network, even if it is compressed a lot through the next layer.

References

[1] F. Pedregosa et al. *sklearn.feature_extraction.text.HashingVectorizer*. Last updated: 29 Mar 2020, (Used 30.03.2020). URL: `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html`.