Intelligent Agents Assignment 1

Erik Norlin

February, 2023

Problem 1.1

Code for an n-gram language model was implemented for extracting and analyzing n-grams of a data set consisting of spoken and written language. The relationship between written and spoken language was also analyzed by looking at the ratio of occurrences between shared tokens (i.e. $r = n_w/n_s$, where n_w and n_s are the number of instances, of a shared token, in the written and spoken set respectively), in hope to gain insight about potential differences or similarities between written and spoken language in the data set.

The n-gram language model was implemented in C# .NET Windows Forms, by first separating the data into two different data sets of spoken and written sentences respectively, following by tokenizing the data by splitting all sentences at appropriate characters. Dictionaries were created and sorted alphabetically for each data set, containing all tokens and their number of instances from respective data set. Indices, corresponding to the alphabetical order of the dictionaries, were assigned to each token of every sentence, allowing reconstruction of each sentence by index referencing to the dictionaries rather than searching through list of strings, minimizing computational cost. As for the n-grams; unigrams, bi-grams and tri-grams were then generated for each data set, and then counted the number of instances of each n-gram. Next, the ratio $r = n_w/n_s$ was computed. Lastly, displaying information about n-grams, tokens and occurrences for the written and spoken data set respectively, also showing shared tokens and the ratio, $r = n_w/n_s$, giving us insight about typical patterns for written and spoken language in the data sets.

Before we dive into the discussion about the differences between the the written and spoken language of the two data sets, it's worth mentioning what the ratio, r, really means. It's quiet simple, large r-value of a token means that it's more prevalent in the written text and small r-value means that it's more prevalent in the spoken text. From the result we can observe that tokens with large r-values tend to be more of things (nouns), years, locations, amounts, what things were, and verbs of past tense, i.e. what happened. On the other hand, tokens with small r-values tend to be more of words one would use in every day life, *"socialising words"*, (e.g. "thanks", "yeah", "yesterday"), what things are, what people have, what people will do, verbs of the future. i.e. what is going to happen. There are however some tokens with high and low r-ratio that are difficult to interpret what they are supposed to represent, here we neglect them. From this we can conclude that in this data set, written language is more oriented towards things and the past, where as spoken language is more oriented towards people and the future.

Problem 1.2

Code for a (binary) Bayesian text classifier was implemented to classify positive and negative restaurant reviews. The classifier was implemented in C# .NET Windows Forms, by first pre-processing the data, which involved cleaning, tokenizing and removing stop words. Following by estimating prior probabilities, $\hat{P}(c_j)$, for each class, 0 and 1 (c_0 and c_1); negative and positive reviews. Next, conditional word probabilities, $\hat{P}(w_i|c_j)$, were estimated for every word in the training set, where this probability is the probability that a particular word in the training set occurs in a given class. Lastly, the training set along with a test set (that the classifier wasn't trained on) were classified, measuring performance of the classifier at the end.

(a) Table 1 shows the estimated prior probabilities from the training set.

Table 1: Estimated probabilities of a document belonging to class 0 or 1 based on the training set, without any further information than the given labels of each document. $\hat{P}(c_0)$ is the estimated probability that a document is a negative review (class 0), and $\hat{P}(c_1)$ is the estimated probability that a document is a positive review (class 1). As can be seen based on the training set, there's a greater probability that a given review, without any further information, is a positive review.

Prior probabilities	
$\hat{P}(c_0)$	0,456
$\hat{P}(c_1)$	0,544

(b) Here we analyze the posterior probabilities that a document belongs to each class for a few words, *"friendly"*, *"perfectly"*, *"horribly"* and *"poor"* respectively. For this we use *Bayes' rule* as followed.

$$\hat{P}(c_j|t) = \frac{\hat{P}(c_j)\hat{P}(w|c_j)}{\hat{P}(w)}, \text{ where } w = t. \tag{1}$$

We simply look at the output of the classification, put the probabilities into eq. 1 and get the posterior probabilities for each word, which can be seen in table 2

Table 2: Estimated posterior probabilities for a couple of words (tokens) from the training set. $\hat{P}(c_0|t)$ is the estimated probability of classifying a negative review (class 0), and $\hat{P}(c_1|t)$ is the estimated probability of classifying a positive review (class 1), given token t. Positive words give higher posterior probability for class 1, and negative words give higher posterior probability for class 0.

Token, (t)	$\hat{P}(c_0\|t)$	$\hat{P}(c_1\|t)$
friendly	0.0900	0.5839
perfectly	0.1799	0.8240
horrible	0.6742	0.1545
poor	0.6473	0.1236

From looking at table 2 we can see that the classifier shows that the positive words, i.e. *friendly* and *perfectly*, are significantly more likely to be classified as positive reviews, and the negative words, i.e. *horrible* and *poor*, are in a similar way significantly more likely to be classified as negative reviews. This is what one would expect, that positive words are more associated with positve reviews and negative words are more associated with negative reviews.

(c) The performances of both the training and the test set were measured after classification of the two were done. From table 3 we can see that the performance of the training set is significantly better than the test set, and that the performance of the test set isn't that good. Reasons for this could be due to (1) too small training set, making it difficult for the classifier to classify unseen data. (2) Too large test set, meaning that even if the training set is large, the test set may be too large in proportion to what it can classify unseen data, which implies the first reason. (3) Too large training set leading to over-fitting, i.e. the classifier could be trained to the noise of the training set, making it difficult to classify unseen data. (4) Removal of stop words could be removal of important information, or on the other hand, not enough removal of stop words to separate the important information from the noise.

Table 3: Performance measure of both the training and the test set. The training set shows better performance than the test set.

Performance measure		
Data set	Training	Test
Precision	0.9850	0.6667
Recall	0.9632	0.8780
Accuracy	0.9720	0.7700
F1	0.9740	0.7579

Problem 1.3

Code was implemented to generate an auto-complete function as used, for example, when writing text messages or an e-mail. The auto-complete function was implemented in C# .NET Windows Forms, by extending the code from the n-gram language model created and discussed about in section 1.1. A text-box was added for the user to write text as they wish, and suggestions of words for the next word show up in a list-box based on matched n-grams to the user input, where the last token of a matched n-gram is the auto-completion. These suggestions are generated bi-grams and tri-grams of the spoken language from the same data set as in section 1.1, sorted in descending order of frequency of occurrence in the data set as the suggestions appear to the user. When the user types something in the text-box, the program will first look for matching tri-grams for auto-completion and only consider bi-grams if no matching tri-grams are found. If no matching bi-grams are matched either, no auto-completion is shown at all to user.

Implementing the function for auto-completion was certainly tricky because of how many if-scenarios one has to consider when looking for matching tri-grams or bi-grams to the user input. Also because one has to be aware of noise in the data, e.g. considering faulty

tokens and hence managing strings, which can be tedious. An example is that *"calculatoryes ..."* should not appear as a matched bi-gram if the user types *"yes"* into the text-box.

The program is fairly slow at generating the bi-grams and tri-grams for the spoken set, but that could mainly have to do with that the data set that's used here is large enough to cause performance issues. However, the performance is not so great when it comes to show. The program lags when using the auto-complete function for the large data set. This is expected considering the large amount of n-grams the program have to run through every time the user writes or deletes a word. The n-grams go through a lot of programming logic to make the auto-completion possible, which can have a large effect on how smoothly the program works. In addition to support this statement, the program runs smoothly on the smaller data set. To conclude, the auto-complete function *itself* works as expected, free from bugs that can be observed. The code however, could be improved to make the program run smoother for the user.

Problem 1.4

BERT (Bidirectional Encoder Representations from Transformers)

(Q1) Code was added to *Google Colab classify_text_with_bert* to explore the average length (number of tokens) of the movie reviews of the test set after BERT's tokenization. It was discovered that the average number of tokens of these reviews are approximately 123.3 tokens per review.

(Q2) The model that will be described in this section is the original uncased BERT-model even though a smaller BERT-model was used in the analysis. The intention was to run the original BERT-model that will be explained, but due to performance issues running this large model, compromises had to be done, thus, a smaller version of BERT was run instead for the analysis. The original model compared to smaller BERT-models works a bit differently, but conceptually very similar. Hence, explaining the original model gives a fundamental conceptual understanding of how BERT works.

The original BERT-model uncased, uncased meaning that the pre-processing of the text sets the whole text to lower case. Fig. 1 shows how this model works [1].

Before text is fed into the model the text must be pre-processed. This includes cleaning of the data, setting the text to lower case and tokenization. The start-token of every segment is the [CLS]-token, which plays an important role in classification which will be described in more detail later.

When text has been pre-processed, each token is *embedded* into a vector of 768 dimensions based on dictionary token indexing, masking and segment type that the text has been categorized into, where an embedding is a feature representation of a particular token. Dictionary token indexing is a way of storing the information about tokens from the input in a more efficient way than storing literal token strings. Storing and handling strings would make the model much slower than handling integers (token indices). Segment type is a way of storing information about which tokens belong to what sentence, and the

purpose of masking is to mask random words in order to train the model to fill in the gaps with the correct tokens, where the masking has been added [1].

After the text has been embedded, the embeddings are sent into the BERT-model consisting of stacked transformer blocks (only encoders). Each encoder consists of a self-attention block and a feed forward neural network (FFNN) where the purpose of self-attention is to give more attention to tokens that play a greater role in the context of the sentence. After the word embeddings has passed through all encoders the output comes out in a form called the *"last hidden state"* consisting of high dimensional vectors that are representations of each token. The last hidden state is either sent to classification or mass language modelling, depending on the task at hand. For classification tasks, only the state vector of the [CLS]-token is interesting because as a segment is sent into and being processed by the pre-trained BERT-model, multiple attention heads will force the [CLS]-token to become dependent on all other tokens in the segment. The [CLS]-token hence becomes an appropriate representation of the context of a given segment [1], [4].

For classification, the [CLS]-token is sent into a number of additional layers that eventually classifies the text input (see fig. 2). These layers are explained in more detail in section (Q5).

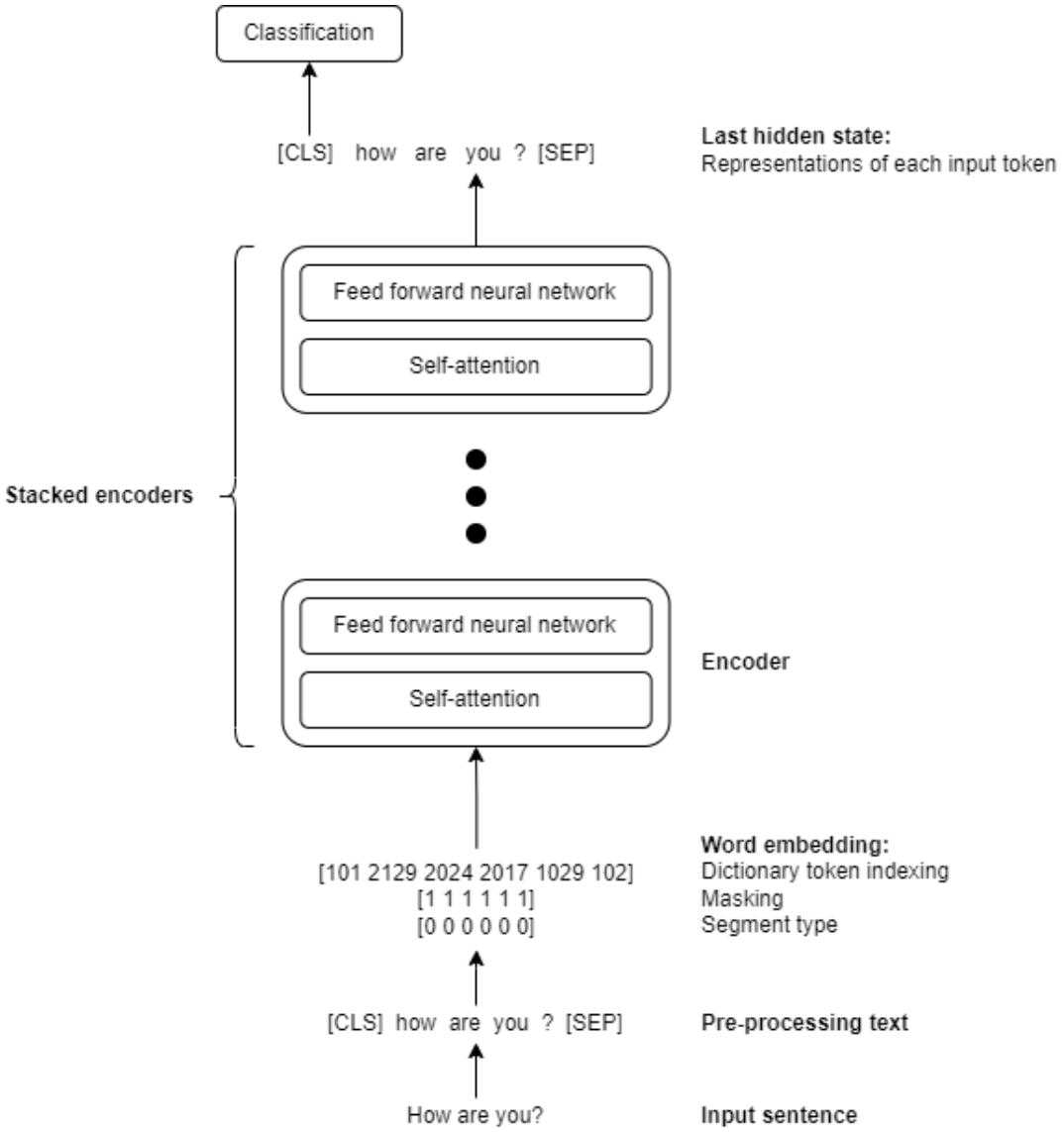

Figure 1: A flowchart and overview of how the original uncased BERT-model works. The input text is first pre-processed, then embedded, following by being sent into a stack of encoders. The output of the model, i.e. the last hidden state is then further processed for either classification or mass language modelling, classification in this case.

(Q3) `"input_word_ids"` is a vector containing the information about the tokens from the input, more precisely, the indices of the corresponding tokens in BERT's dictionary, i.e. token indices, as previously mentioned. Let's demonstrate a sentence *"How are you?"* in token indices from the real BERT. `input_word_ids` of this is [101 2129 2024 2017 1029 102].

`"input_type_ids"` is a vector containing the information about which tokens belong to what sentence. This vector of the previous example is [0 0 0 0 0 0], all types being the same since all tokens belong to the same sentence.

`"input_mask"` is a vector containing binary values that masks some of tokens of the input with boolean operation. Masking of the previous example could be where `input_mask` is [1 1 1 1 1 1], meaning that no tokens are masked here due to the 1s being "true" for all

tokens in the sentence.

(Q4) When the embedded input has passed through the model, the last hidden state vector of the [CLS]-token is then passed through a pooling layer consisting of a linear layer with a tanh activation function. The output from this pooling layer is the `"pooled_output"`, still having a vector length of 768. So to answer the question, `pooled_output` grabs the state vector of the [CLS]-token after it having passed through a pooling layer.

(Q5) For text classification (see fig. 2), the last hidden state is sent through a pooling layer as previously mentioned, and the output being `pooled_output` is then sent to a *dropout* layer. The idea behind dropout is to avoid over-fitting of the model during training. This is done by temporarily, and probabilistically *dropping out* a node in the network along with its' weights connecting to other nodes to avoid weights from getting too much training. Temporarily as some weights are dropped out, gives the network different structure, and this forces the network to learn without getting too dependent on certain weights. This has the effect of adding *noise* to the data that the model is trained on, and thus, helps avoiding over-fitting. The dropout layer is followed by a linear layer that classifies the text, in this case we're dealing with binary classification [1], [4], [3].

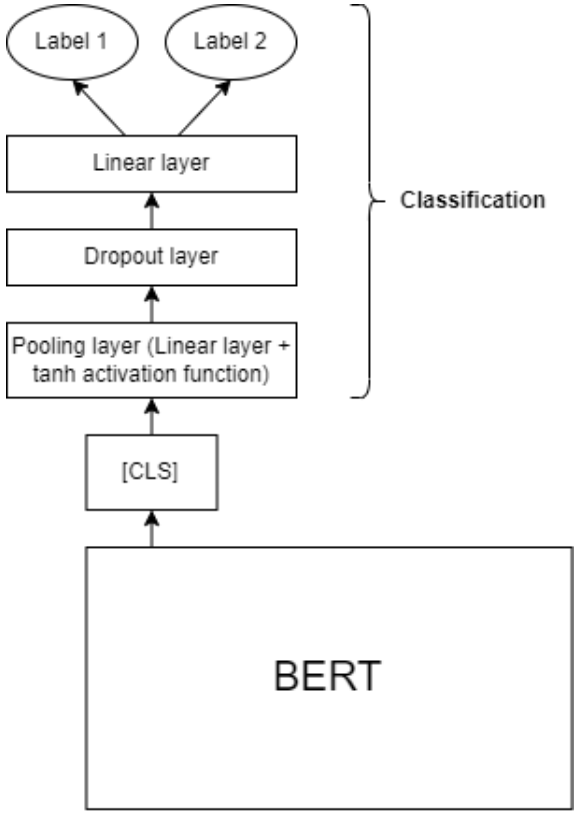

Figure 2: Rough flowchart of how the classification layers works in BERT. The state vector of the [CLS]-token is passed into a pooling layer consisting of a linear layer along with a tanh activation function. The output of this is `pooled_output` which is passed into a dropout layer and then into a linear layer. Finally, the linear layer classifies the sentence that was fed into BERT in the beginning. In this case we have a binary classification.

(Q6) *AdamW* is a stochastic gradient descent method and a state-of-art optimizer for neu-

ral networks based on the widely used *Adam* optimizer, where an optimizer is a method for adjusting the weights of a model during training. The special thing about AdamW and how it differs from Adam is that it uses uncoupled weight decay, which gives substantially better generalization performance than the Adam optimizer [2].

Benchmark model

A benchmark model was implemented to compare the performance of BERT when classifying negative and positive movie reviews. The implementation was done in C# .NET Windows Forms, by extending the code from the n-gram language model in section 1.1, as well as the Bayesian classifier in section 1.2. The BERT model that was analyzed was one of the smaller BERT model, more precisely
`"small_bert/bert_en_uncased_L-4_H-128_A-2"`. Similarly to what was done in section 1.1 the ratio between shared tokens of the two classes/data sets was evaluated, though this time with r as $r = log_{10}(n_+/n_-)$, where n_+ is the number of instances of a shared token between the classes in the positive reviews, and n_- is the number of instances of the same token in the negative reviews. Table 4 shows the 30 most positive values of r, and table 6 shows the 30 most negative values of r.

One can see in table 4 that the tokens with the 30 largest values of r are arguably uncommon words. One hypothesis is that these words are names of well known professionals within the industry that are known to produce better movies, and these words would therefore be associated with positive reviews. Another word *"excellently"* shows up in this list, indicating that positive words are more common in positive reviews, as expected. Similarly goes for the top negative values of r. We can also see in table 6 that uncommon words, arguably names, are associated with worse movies, which also makes sense. Negative words such as *"awfulness"*, *"unwatchable"*, and *"appallingly"* are on this list as well, indicating that negative words are associated with negative reviews, which is also expected. It's also possible that there's noise in the data causing some tokens being perhaps unexplainable. The way of cleaning and tokenizing of the data could possibly be partially responsible this when implementing the benchmark model. Also, the data set could have inherit noise from the beginning.

Table 4: Tokens with the 30 most positive values of r.

Token	r
paulie	2
iturbi	1.68
gundam	1.64
philo	1.63
feinstone	1.61
giovanna	1.6
lindy	1.6
rosenstrasse	1.57
anchors	1.56
clutter	1.56
fassbinder	1.52
luzhin	1.52
callahan	1.51
victoria's	1.51
aiello	1.49
stevenson	1.48
excellently	1.47
aviv	1.46
hickock	1.46
mathieu	1.46
conroy	1.45
dench	1.45
korda	1.45
partition	1.45
ashraf	1.43
emil	1.43
jabba	1.43
hayworth	1.42
askey	1.41
felix	1.41

Table 5: Tokens with the 30 most negative values of r.

Token	r
point-	-3.27
boll	-2.11
uwe	-2
thunderbirds	-1.79
beowulf	-1.75
wayans	-1.67
ajay	-1.66
seagal	-1.66
dahmer	-1.62
awfulness	-1.58
steaming	-1.57
grendel	-1.56
segal	-1.53
deathstalker	-1.52
stinker	-1.51
interminable	-1.48
forwarding	-1.46
sabretooth	-1.45
gamera	-1.43
devgan	-1.41
dreck	-1.41
picker	-1.41
unwatchable	-1.41
razzie	-1.4
nada	-1.38
mst3k	-1.37
nostril	-1.36
aag	-1.34
appallingly	-1.34
demi	-1.34

Performance measure was carried out for both BERT and the benchmark model on the test set of the movie reviews. In table 3 we can see that the performances of both models does not differ very much from each other. What is interesting is that accuracy, recall and F1 for the benchmark model shows better results than BERT, even though BERT is a much more complex model. However, it is worth mentioning again that the BERT model used here was a smaller version of the original BERT. A larger BERT model would perform better. Either way, this shows that it is not always better to use a black box model before an interpretable model like a Bayesian classifier used here, which should be noted. Black boxes are hyped up today, and they are not better just because they are more complex, as it shows here.

Table 6: Performance measure from the implemented benchmark model and the smaller BERT model, `small_bert/bert_en_uncased_L-4_H-128_A-2`, on the test set of movie reviews.

Model	Accuracy	Precision	Recall	F1
Benchmark	0.8252	0.8117	0.8469	0.8289
BERT	0.8118	0.8688	0.7063	0.7792

References

[1] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[2] Ilya Loshchilov and Frank Hutter. "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101* (2017).

[3] TensorFlow. "https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout". In: (2022).

[4] BERT (Bidirectional Encoder Representations from Transformers). "https://github.com/tensorflow/models/tree/master/official/legacy/bert". In: (2019).