Intelligent Agents (TME286)
Assignment 1: Text processing and language models

Problem 1.1
$n-$gram language models (10p, mandatory)

In this problem you will implement code for extracting and analyzing $n-$grams. You will also study the difference between written text and (transcripts of) spoken text. For this problem, you will use a data set containing both written text (from Wikipedia) and transcripts of spoken text (containing utterances from an American radio show). In this case, a lot of hard work has been done for you: The data file contains lists of sentences, where most special characters have already been removed. However, there may be some errors remaining in the data.

Your program code should be written in C#, starting from the base code in the folder `Problem1.1/` contained in the file `StartingPointAssignment1.zip`. That file contains skeleton code for a C# program that can read the data file (`SpokenWritten.txt`), which is available (large file!) on the Canvas page for Assignment 1. There is also a smaller data set, `SpokenWrittenSmall.txt` (with a subset of the sentences from the full data set), which you can use when testing your code. *Note, however, that the analysis that you hand in should be based on the full data set, in the file* ***`SpokenWritten.txt`****.*

You need to add methods for tokenization, indexing and dictionary generation, as well as further ($n-$gram) processing. You should do the following:

Tokenization

Write the Tokenize() method (on the level of sentences) that will split sentences into tokens (words, mostly, but the list of tokens will also contain numbers, compound words, contractions etc.), all in lower case. Note that the tokenization should remove end-of-sentence characters (e.g. full stops) as well as commas between words. However, be careful not to remove (for example) full stops that do not represent the end of a sentence (e.g. in abbreviations). Do not split contractions, i.e. maintain words like *i've*, *they're* and so on as a single token.

Hint: Use the Split(), Trim(), and TrimEnd() commands (that are applied to strings) in C#.

Indexing and dictionary generation

Here, you should make dictionaries (one for the written data, and one for the spoken data), consisting of items that each represent a given token but also the number of instances of that token, in the data set under consideration. The dictionaries should be sorted in alphabetical order. You should then find and assign token indices to all tokens in every sentence. Note that, in this case, the token index lists (one for each sentence) are not really needed, but it is a good exercise to learn how to generate them (and it may be useful in later assignments). You will, however, need the dictionary below, when looking for frequent unigrams (=tokens).

Processing

After the two preliminary steps above, write code (that will be called via the Process button event handler) for (i) finding the 300 most frequent 1-grams (from the dictionaries; see the previous paragraph), 2-grams, and 3-grams in each data set, (iii) the number of distinct unigrams in the written and spoken sets (i.e. the number of items in each dictionary), as well as the number of shared tokens, i.e. tokens that are present in both the spoken and written sets. (iii) among the shared tokens (i.e. those that are present in *both* the written and the spoken data set), find the 50 tokens with the largest values, and the 50 tokens with the smallest values, of the ratio $r = n_w/n_s$, where n_w and n_s are the number of instances (of a given token) in the written and spoken sets, respectively. The entire output from the analysis just described should be printed to the text box in the main form (clearly and neatly formatted, with appropriate headers so that the reader can easily follow the analysis). You should also save the contents of the text box to a file (Analysis1.1.txt). Finally, you should analyse the results just found, giving some insightful comments regarding the differences between written and spoken language.

Hint Since the lists of $n-$grams will be very large, before sorting them (to find the most frequent ones), run through each NGramSet instance and remove all items for which the number of instances is small (below 3, say). This is a linear process, which will significantly speed up the next step, where you need to sort the NGramSet based on the number of instances of the remaining items.

Notes
(1) Detailed instructions regarding the various required steps can be found as comments in the C# base code.
(2) Since the data sets are large, you must probably run your C# code in 64-bit mode. See the instructions in the FAQ (on the course web page).

What to hand in Write a section (1.1) in your report, where you briefly describe your implementation, as well as the analysis of the difference between spoken and written language. Also, hand the file Analysis1.1.txt described above. You should hand in your C# program code (the entire solution, compressed as a zip (or 7z) file).

Evaluation For this problem, the implementation is worth 7p and the report 3p. If you need to resubmit the problem, a maximum of 6p will be given. If the program code does not work (directly, without any editing), the problem will be automatically returned and resubmission will be required.

Problem 1.2
Bayesian text classification (10p, mandatory)

Here, you will implement a (binary) Bayesian text classifier for restaurant reviews. The training set, contained in the file `RestaurantReviewsTrainingSet.txt` (available in the `Data/` folder; see below) contains 500 restaurant reviews, some of which are positive (label 1) and some that are negative (label 0). There is also a test set in the file `RestaurantReviewsTestSet.txt`, with 100 restaurant reviews.

You should start from the C# base code in the folder `Problem1.2/` contained in the file `StartingPointAssignment1.zip` (on the course web page) and then implement the following: (1) preprocessing (removing special characters, normalizing, and so on), (2) tokenization (without lemmatization or stemming), and (3) stop word removal (using the list `StopWords.txt`, available in the `Data/` folder). Detailed instructions are given as comments in the C# base code. Next, you should implement a Bayesian text classifier, following the method described in Sect. 4.2 of the compendium. Here, too, you will find detailed instructions as comments in the C# base code.

Once you have completed the implementation, run it over the training set and assign class labels using maximum a posteriori probabilities (as in the compendium), using the label 1 for positive reviews and 0 for negative reviews. Finally, apply the classifier (without changing it) to the test set. In this step, any token (word) t that is absent from the training set should be ignored.

What to hand in Write a section (1.2) in your report, where you briefly describe your implementation. Then (a) write down the prior probabilities for the training set (for each class); (b) write down the two probabilities $\hat{P}(c_0|t)$ and $\hat{P}(c_1|t)$ for the words *friendly*, *perfectly*, *horrible*, and *poor*, and provide a brief analysis based on your findings; (c) write down the performance measures (precision, recall, accuracy, and F1) over both the training set and the test set. Most likely, you will find that the performance over the test set is not that good. Try to explain why.

Furthermore, hand in all your C# program code (the entire solution, compressed as a zip (or 7z) file), as well as the complete analysis in a text file `Analysis1.2.txt` (which can be saved directly from the C# program; all the required code for this step has already been implemented; see the menu item *Save analysis* in the program). You should *not* hand in the data sets - we have our own copies!

Evaluation For this problem, the implementation is worth 7p and the report 3p. If you need to resubmit the problem, a maximum of 6p will be given.

Problem 1.3
Autocompletion with n-grams (10p, voluntary)

In this problem you will extend your results from Problem 1.1 to generate an auto-complete function as used, for example, when writing text messages or an e-mail. Start by copying the *entire* `Src/` folder from Problem 1.1 (including the code that you have added) into a new folder called `Problem 1.3/`. Do *not* attempt to combine the program code for the two problems: For both Problem 1.1 and Problem 1.3 you should hand in separate, complete C# solutions that do not depend on each other in any way. You can rename the solution and the application (for Problem 1.3) if you wish.

Next, remove the main form and generate a new main form, with a menu strip containing a file menu with two items: `Load data` and `Exit`, such that event handler for the first item loads the data (you may re-use code from Problem 1.1) and the second item simply exits the program. Note: In this case, use the same data set as for Problem 1.1, but discard all written data, i.e. retain only sentences from Class 0 (spoken). Then add a top menu item (next to the `File` item) called `Processing`, with a sub-item `Generate 3-grams`. When that item is selected, the program should generate an alphabetically sorted list of 2-grams from the (note!) spoken data set, as well as an alphabetically sorted list of 3-grams from the same set. The alphabetical sorting should be based on the entire string (all tokens concatenated, with space between tokens). You may re-use code that you have written for Problem 1.1.

Next, on the main form, add a split container. In the split container, add a text box (for typing) and a list box (for showing suggestions). The appearance of the GUI should be roughly as shown in the image `AutocompleteGUI` on the Canvas page. Then, write an event handler for the `TextChanged` event in the text box, such that, when the user completes a word (entering a space character), the $n-$gram model will search for suitable 3-grams, using the two most recent words as the first two words in the 3-gram and then presenting (in the list box), the suggestions for the final word of the 3-gram, in falling order of frequency. If no 3-gram can be found that starts with the two most recently typed words, the program should instead resort to 2-grams, using the last typed word as the first word of the 2-gram, and then presenting candidates for the second word in the list box (as described for the 3-gram case). If this process does not generate any candidate either, the list box should be empty. In order to speed up the search process, you should carry out a binary search over the set of 3-grams and 2-grams.

What to hand in Write a section (1.3) in your report, where you briefly describe your implementation. You should also briefly discuss the performance of you autocomplete function. You should hand in your C# program code (the entire solution, compressed as a zip (or 7z) file).

Evaluation For this problem, the implementation is worth 7p and the report 3p. Resubmissions are not allowed.

Problem 1.4
Text classification with BERT (15p, voluntary)

In this problem you will use BERT (on Google's Colab) for classification of movie reviews. If you do not have a Google account, start by setting one up, and then open Colab. Now, for almost any problem involving deep networks in NLP, there are plenty of resources available online, in the form of cookbook-style (step-by-step) sequences of Python instructions. In this case, you can find a full Jupyter notebook, containing all the required steps at `https://colab.research.google.com/github/tensorflow/text/blob/master/docs/tutorials/classify_text_with_bert.ipynb`

Classification with BERT

Your task will be to run through this notebook, selecting a suitable version of BERT for the analysis. However, you should also try to understand what the code does, and also make a simple benchmark classifier for comparison. Start by running through the notebook on Colab, which will train BERT using 20,000 reviews from the training set (keeping the remaining 5,000 for validation). There is also a test set with another 25,000 reviews. In the process of running through the notebook, answer the following questions:

(Q1) What is the average length (number of tokens) of the movie reviews in the test set (25,000 reviews), after BERT's tokenization? (you will need to add some code in the Jupyter notebook)

(Q2) What is the structure of the BERT version that you choose for your analysis? Follow the links under *Loading models from TensorFlow Hub*, read the corresponding scientific paper, and describe the model in as much detail as you can. Include at least one figure, with a clearly written figure caption (that you should write yourself).

(Q3) Describe, in detail, the output of BERT's preprocessing, i.e. the contents of the vectors `input_word_ids`, `input_mask`, and `input_type_ids`. In addition to the general description, provide a specific example, using a sentence of your choice.

(Q4) What does the `pooled_output` (under *Define your model*) do?

(Q5) How do the added layers (for classification) look (i.e. after the pooled output)? Draw a figure and include in your report. Also, explain clearly what the *dropout* layer does.

(Q6) Describe the chosen optimization method (AdamW per default).

Benchmark model

Next, download (outside Colab) the Large Movie Review Dataset, which can be found at `https://ai.stanford.edu/~amaas/data/sentiment/`. Then, writing code either in Python or C# (you may use, to the extent possible, C# code that you have written for the earlier assignments), clean (remove special characters, and so on, as in Assignment1.2) and tokenize the movie reviews, and then compute (over the 25,000 reviews in the original training set, i.e. do not split into training and validation here as was done for BERT) for each token t, the measure $\Gamma(t) = \log(n_+/n_-)$, where log refers to the base-10 logarithm, and n_+ and n_- are the number of instances of a given token (t) in the positive and negative reviews, respectively. For example, if a token t appears in 154 positive reviews and in 37 negative reviews, its values of Γ will be $\log(154/37) = 0.6193$. Be careful to typecast the denominator correctly (in case you use C#), i.e. as a `double`, so that the program does not treat the ratio as an integer division (in which case 154/37 will give 4, rather than 4.162...).

In this case, do *not* use the BERT sub-word tokenizer. Instead, tokenize into words (not sub-words) using, for example, tokenization code that you have written for the earlier assignments. If you choose to write in Python, you must rewrite the tokenization code, but that is quite easy.

Consider only those tokens for which there is at least one instance in both sets (to avoid division by zero and to avoid taking the logarithm of 0!) Ignore all other tokens. Then make a neatly printed list, sorted in alphabetical order, with rows containing the tokens and their respective values of Γ, separated by a tab character. This list should be saved as `TokenList.txt`. (Do *not* include it in the report; it should be submitted as a separate file, see below.)

Next, write a simple classifier as follows: For any given review, sum the values of Γ for all the tokens, ignoring any tokens that do not have a value of Γ. If the sum is positive, classify the review as being positive, otherwise negative. Then compute the accuracy, precision, recall, and F1 measures (over the 25,000 revews in the training set and, more importantly, over the 25,000 reviews in the test set). These numbers can then serve as a benchmark for comparison with BERT's performance.

What to hand in Write a section (1.4) in your report, where you give clear and thorough answers to the six questions above (Q1) - (Q6). Do not copy-paste from explanations found online: Your answers must clearly indicate that you know what you are writing! Next, briefly describe the implementation of the simple benchmark classifier, and list the 30 tokens with the most positive values of Γ as well as the 30 tokens with the most negative values of Γ (all in a neatly formatted table). Also, compare these two lists, and provide some conclusions regarding their content. Then, make a table showing the accuracy, precision, recall, and F1 scores over the test set (25,000 reviews) for (a) the simple benchmark model and (b) BERT. Thus, this table should contain two rows (BERT-test, benchmark-test). You should also provide some comments regarding the performance (difference) between the two models - the simple benchmark model and the very complex BERT model.

You should hand in the file `TokenList.txt` as well as your Python or C# program code (the entire solution, compressed as a zip or 7z file) for the benchmark model, as well as the entire Jupiter notebook for BERT (File - Download ipynb in Colab).
Note: Do not, under any circumstance, dump screenshots in the report. Write the text separately for the report.

Evaluation For this problem, the implementation is worth 8p and the report 7p. Resubmissions are not allowed.