Intelligent Agents (TME286)
Assignment 1: Text preprocessing and classification

1.1 Part-of-speech (POS) tagging (10p, mandatory)

In this problem you will investigate part-of-speech (POS) tagging, by first implementing and evaluating a unigram tagger in C#, and then evaluating an existing Python implementation of a perceptron tagger.

The data set used will be the Brown corpus (available on the Canvas page for Assignment 1 in the file `BrownCorpus.txt`). For Part (a), you should write your own code, starting from the skeleton code provided in the file `Src1.1.zip` available on the web page. This program already contains code for reading and formatting the Brown corpus in a suitable way.

(a) Starting from the C# program just mentioned, add code to the program to do the following (in this order)

1. Read the entire Brown data set (responding to the `File - Load POS corpus` menu item).
2. Reduce the tag set for the Brown data set. Originally, it contains very many (hundreds) of different tags. You should reduce them the 12 universal POS tags, using the mapping available in the file `BrownToUniversalMapping.txt` on the Canvas page for Assignment 1. This is a two-step process: First, responding to the `File - Load tag conversion data` menu item, you should load and parse the mapping file, to generate this mapping in a suitable format. Next, responding to the event handler for the button marked `Convert POS tags (Brown` $\rightarrow$ `Universal)`, you should carry out the actual conversion.
3. Split the data, with the first 80% of the sentences in the Brown data set being placed in the training set, and the remaining 20% in the test set. (A validation set is not needed here, since no optimization will be carried out).
4. Write code for generating basic statistics for the (note!) training set, in the event handler for the button marked `Generate statistics`. Your program should do the following: (i) Count the number of instances of each POS tag, and compute the corresponding fractions (i.e., dividing by the total number tagged words), (ii) count the fraction of words that are associated with $1, 2, \ldots$ different POS tags. For example, the word *organization* has only one tag, NOUN, whereas the word *bear* has two, NOUN and VERB, and so on.
5. Write code for generating a unigram tagger (in the event handler for the button `Generate unigram tagger`, i.e., a tagger that simply assigns the most common tag for a given word. The unigram tagger should be generated using the *training* set.
6. Write code for running the unigram tagger over the (note!) *test* set, in the event handler for the `Run unigram tagger` button. The output should be the accuracy of the tagger (see Chapter 4, Section 4.6).

In the cases that generate output (i.e., Steps 4 and 6), the output should be written (in a neat format) to the list box (`resultsListBox`) on the main form (window) of the application.

(b) In this part of the assignment, you should use Google Colab to evaluate the perceptron tagger; see also the example in Appendix B in the compendium. Write a Jupyter notebook in Colab, using the two text files `BrownCorpus.txt` and `BrownToUniversalMapping.txt` as data, to do the following:

1. Import the standard (predefined) perceptron tagger (available in the NLTK library), as described in Appendix B.

2. After uploading (in Colab) the data file mentioned above, re-format the sentences in the Brown set to generate a suitable data structure, with words and their associated POS tags, which you should convert to the Universal POS tag set, as in Part (a). **Note:** make sure also to convert all words to lowercase, as in (a); see also the note below.

3. Split the set into training and test sets precisely as in Part (a) above. You will only need the test set here, since you will use a pre-trained POS tagger.

4. Tokenize the test set (by simply splitting it based on the space character).

5. Run the perceptron tagger over the test set.

6. Map the inferred tags (obtained with the perceptron tagger in the previous step) to the Universal POS tag set. Note that the perceptron tagger has been trained using the Penn treebank POS tag set rather than the Brown tag set. Thus, you must use the `map_tag` function in NLTK (see `https://www.nltk.org/api/nltk.tag.mapping.html`) to map the (Penn treebank) tags (that you obtain from the perceptron tagger) to the Universal tag set, using `'en-ptb'` as the source tag set.

7. Compute the accuracy of the perceptron tagger over the test set. Since both the ground truth tags and the inferred tags have been mapped to the Universal POS tag set (in the previous steps), the results obtained will be directly comparable to those obtained for the unigram tagger above.

Notes For simplicity, the words are converted to lowercase when the data set is loaded in the C# program. This leads to some (small) loss of information, but that is not important here.

What to hand in You should hand in (i) the *entire* C# solution from Part (a). The corresponding application should be possible to compile and run directly, producing neatly formatted output as described in Part (a); (ii) the entire Jupyter notebook from Part (b), including all steps; (iii) a section (Called *Problem 1.1*) where you briefly describe your C# implementation, provide the statistics computed in Step 4 of Part (a) as well as the results obtained in Step 6 of Part (a) and in the final step of Part (b). You should also include a brief discussion of your findings.

Evaluation For this assignment, the coding (C# and Python, via Colab) is worth 7p and the report 3p. If a resubmission is required, a maximum of 6p will be given.

1.2 Perceptron text classifier (10p, mandatory)

In this problem you will implement a (binary) perceptron text classifier, and will use it to classify a set of airline reviews. You should start from the C# skeleton code in the file `Src1.2.zip` (available on the Canvas page) do the following

1. Write code for tokenizing the data sets. Here, you should write your own tokenizer, which should split the texts into individual words (and punctuation symbols, such as full stop (.), comma (,), question marks (?) and so on). You must be careful to handle (i) abbreviations (using lookup table that you must define; for example, abbreviations such as *e.g.*, *i.e.*, *mr.*, *mrs.*, and so on, should be kept as they are, without any splitting) and (ii) numbers (for example, *3.14* should define *one* token, without being split). Writing the tokenizer *will* take some effort. It is *not* sufficient just to split the sentences using the space character. There is a class `Tokenizer` (in the NLP library) that you should use as a starting point. The tokenization process should be started when the user clicks on the `Tokenize` button. As a by-product of the tokenization, you should also generate a vocabulary, i.e., a data structure (for example, a simple list or an instance of `Dictionary`) that contains the entire vocabulary, in alphabetical order, for the (note!) training set. Thus, all three sets (training, validation, and test) should be tokenized, but the vocabulary should be generated using only the training set.

2. Write code for indexing the data sets. Here, you should generate a data structure that, for each text in each data set, generates a list of word *indices*. In this process, you should use the training set vocabulary (from the previous step), to identify the index (in the vocabulary) of each word in every text, in each set. Words that do not appear in the vocabulary should be assigned index -1, meaning that they will be ignored in the classifier (see below). This will not happen for the training set, since the vocabulary was indeed built using this set, but it will happen for the validation and test sets, which contain at least a few tokens that are not present in the training set.

   The purpose of the indexing is to speed up the computation of the output y of the perceptron classifier, without having to search the vocabulary. Once the texts have been indexed, for any given text, the output y can be obtained by simply summing the weights (in the classifier) whose indices correspond to those in the indexed text. For example, a in a vocabulary with (say) 50,000 words, a sentence such as *I saw a zebra* may be indexed as (13241, 35412, 156, 48941). When computing y one can then simply form the sum of the weights w[13241], w[35412], ... Here, any word with a negative index (i.e., -1; see above) should simply be ignored.

   The indexing (for all three sets) should occur when the user clicks on the `Generate indexed sets` button.

3. Write code that implements a perceptron classifier as well as the corresponding training method (here referred to as the (perceptron) optimizer), as described in Chapter 4. For a training set vocabulary with v words, the perceptron classifier should thus have v weights (and a single bias term), which you may initialize randomly (there are better ways to initialize the classifier, but random initialization is acceptable here).

When the user clicks on the `Initialize optimizer` button, the perceptron tagger should be initialized and the optimizer should be set up (i.e., an instance of the corresponding class should be generated). When the user clicks on the `Run optimizer` button, the optimizer should start (or resume, if it has been stopped, with the `Stop optimizer` button). As the optimizer runs, for each epoch it should (i) first generate a random permutation of the data set in question, (ii) run through each item in the permutation, classifying with the perceptron classifier and updating the weights as described in Eq. (4.10) in the compendium and (iii) (after running through the entire epoch, using the weights then obtained) run through both the training and the validation sets, and compute the classifier's accuracy (over all texts in the respective sets) for both sets, without (at this point) modifying any weights, i.e. using the weights obtained after Step (ii) above. If the (note!) *validation* accuracy exceeds the previous best value, then store the corresponding classifier for later use. This process should then be repeated, epoch by epoch, until termination. The program should run indefinitely until you stop it manually by clicking the `Stop optimizer` button.

4. Then run the training algorithm over the training set (also measuring validation accuracy as described above). Once the training has been completed (and *only* then) you should measure also the accuracy over the test set (as obtained from the best classifier, i.e., the one with highest validation accuracy).

Note Here you *should* use threading, as described in Appendix A.4, to avoid that the GUI freezes during optimization. Note also that you must make sure that any GUI action (e.g., printing to screen) is thread-safe; see also Appendix A.4. For every iteration, print the training and validation accuracy (on the screen).

Hint For the optimization process, it often helps to define (and to use in the optimizer) an evaluator class, e.g., `PerceptronEvaluator` (or something like that) for keeping track of the results obtained when evaluating the perceptron over an entire data set.

What to hand in You should hand in (i) the *entire* C# solution. The corresponding application should be possible to compile and run directly, producing neatly formatted output; (ii) a section (Called *Problem 1.2*) where you briefly describe your C# implementation. You should also include a plot showing the training and validation accuracy for each epoch. Moreover, you should include a table showing (numerically) the training, validation, and test accuracy for the best classifier (i.e., the one with highest validation accuracy). Furthermore, you should include a discussion of your findings, giving some examples (from the test set) of both correctly and incorrecly classified sentences. You should also list the 10 most positive words (i.e., the ones with the largest positive weights) and the 10 most negative words (largest negative weights) in the best classifier.

Evaluation For this assignment the C# program is worth 7p and the report 3p. If a resubmission is required, a maximum of 6p will be given.

1.3 Bayesian text classifier (10p, voluntary)

Here, you will implement a (binary) Bayesian text classifier for restaurant reviews. The training set, contained in the file `RestaurantReviewsTrainingSet.txt` (available on the course web page) 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# skeleton code in the file `Src1.3.zip` (on the Canvas page), and then implement Bayesian text classification (in a new class `NaiveBayesianClassifier`) as described in Chapter 4 of the compendium and in Lecture 4. This is a voluntary assignment, so you will need to do a bit more work yourself than in the two mandatory assignments above: Here, the skeleton code only loads the data sets. Make sure to study carefully the example in Lecture 4 before starting to implement the classifier. Note that, for the Naïve Bayesian classifier, there is no optimization. Thus, here we do not need a validation set: You only need a training set and a test set. Note also that, for this classifier, you may wish to use a different interface for the `Classify` method. See the base class `TextClassifier` for more information.

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 (called *Problem 1.3*) in your report, where you briefly describe your implementation. Then, for the restaurant reviews (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; (d) since the data format is the same as for Problem 1.2 above, you can *also* run the Bayesian classifier over *that* set. Thus, train another Bayesian classifier (same program code, of course) over the airline reviews training set (from Assigment 1.2) and then evaluate it over the corresponding *test* set (not the validation set). Then, in your report, compare the accuracy obtained with the Bayesian classifier to that obtained with the perceptron classifier, and briefly discuss your findings.

Furthermore, you must hand in all your C# program code (the entire solution). You should *not* hand in the data sets - we have our own copies!

Evaluation For this problem, the implementation is worth 6p and the report 4p. Resubmissions are not allowed.

1.4 Autocompletion with n-grams (10p, voluntary)

In this problem you will generate an auto-complete function that can be used, for example, when writing text messages or e-mails.

For this problem, you should generate your *own* data set, by collecting text from the internet. Some examples of suitable sources (for example, Wikipedia, Project Gutenberg, NPR, and so on) can be found on the course Canvas page (under Assignment 1). Your data set should contain at least 5,000,000 tokens. Note: When you collect text data, make sure to respect any rules that may apply regarding automated downloading! Note that you may have to do some preprocessing to get the data in a suitable format (i.e., plain text), a process that, moreover, may differ between different sources.

You should start from the (almost empty) C# skeleton code in the file `Src1.4.zip` (available on the Canvas page). Then, using your text data set, generate the set of unigrams, the set of bigrams, and the set of trigrams, where each n-gram is represented by an identifier (a string containing the words in the n-gram) list of strings (i.e., the individual words in the n-gram), as well as their frequency in the data set. You may use the `NGram` class available in the NLP C# library. Each set (unigrams, bigrams, and trigrams) should be sorted alphabetically based on the identifier. Make sure to include punctuation marks in the n-grams as well (for simplicity, not to have to split the text into sentences).

Your program should do the following: After loading the data set and generating the unigram, bigram, and trigram sets, when a user enters a word in a text box, you should use the trigram set to find the most likely next word (based on the two last words entered so far; if the user has only entered one word, use the trigram that starts with a full stop and is followed by the word entered so far), and then display these options in a list box (both the text box and the list box are available in the skeleton code). If the user hits the tab key, the top option should be chosen, but the user should also have the possibility to just continue entering text in the text box (i.e., even words that are not among the options shown in the list box). If, for a given sequence of two words, no corresponding trigram (one whose first two words are the ones just mentioned) is available, use the bigram set as a fallback option, displaying the frequency-sorted list of likely next words based only on the last word entered. If there is no bigram available either, then just show an empty list box.

What to hand in Write a section (called *Problem 1.4*) in your report, where you briefly describe your implementation. You should also hand in your C# program code (the entire solution), as well as the data set. Note that the entire file (for Assignment 1 as a whole) should preferably not exceed 50 Mb. Please use `7-zip` if possible to make the compressed file as small as possible.

Evaluation For this problem, the implementation is worth 6p and the report 4p. Resubmissions are not allowed.