CS 4740: Intro to NLP
Project 1: Language Modeling

Part 1 due electronically via CMS by Monday 9/12/16, 11:59pm
Part 2 due electronically via CMS by Friday 9/23/16, 11:59pm

1 Overall Goal

In this project we will ultimately explore two applications of n-gram language models: document classification and spelling error correction. You will work up to this via the pieces described in Sections 2–5 below.

You should work in groups of 2 or 3 students. Students in the same group will get the same grade. Make sure that everyone contributes to the project and, at the end of the report, please explain how you divided the tasks between group members (e.g. what part was coded/designed by whom). Please **do not** forget to form groups on CMS. You can use Piazza to find partners.

The report is important! The writeup is where you get to show us that you understand not only *what* you are doing, but also *why* and how you are doing it. So be clear and concise; avoid vagueness and verbiage. (More on this at the end.)

2 Unsmoothed n-grams

To start, you will write a program that computes *unsmoothed unigram and bigram probabilities* from an arbitrary corpus. You can use any programming language(s) you like.
Assume that you are given raw text as input (see 2.1). You may need to do tokenization, based on the design decisions you make. You may use existing tools just for the purpose of preprocessing (e.g. word and sentence tokenization) but **you must write the code for computing n-gram probabilities yourself.** For example, consider the simple corpus consisting of the sole sentence:

the students liked the assignment

Part of what your program computes could be the following:

```
...
P(the) = 0.4
P(liked) = 0.2
...
P(the|liked) = 1.0
P(students|the) = 0.5
...
```

2.1 Dataset

You are given (via CMS) a collection of news articles from the *20 newsgroups* dataset.[1] In the data folder, you will see various folders including training documents for different topics (atheism, autos, religion, medicine etc.) You should consider each topic folder as a **single corpus**, and generate a language model for each. (But feel free to explore other ways to combine the topics if you can justify why, preferably with numbers.)

2.2 Preprocessing

The files are given in raw text format, so you will need to perform sentence splitting and/or tokenization. Moreover, you may want to look into the files and do further preprocessing to remove irrelevant text from them (e.g. in the beginning of each file there is email address information that you may want to remove). You probably will want to add sentence boundary tags. (Keep track of your decisions since you will need to include them in the report.)

3 Random sentence generation

Next, you will write code to generate random sentences based on your unigram and bigram language model(s). Experiment by developing a random sentence generator for each corpus (topic). Also, experiment with seeding (i.e. starting from an incomplete sentence of your choice and completing it by generating from your language model, instead of generating from scratch). Include examples of the generated sentences in your report. Analyze the generated sentences and compare the sentences generated with unigram vs. the bigram models. **Include the generated sentences and your analysis in both Part 1 and Part 2 submissions.**

4 Smoothing and unknown words

Next, you will implement Good-Turing smoothing and a method to handle unknown words in the test data. Explain why smoothing and unknown word handling is important. After finishing the rest of the assignment, think back about smoothing and explain its role in the subsequent applications.

5 Perplexity

Implement code to compute the perplexity of a test set. **Compute and report the perplexity of each of the language models (trained on each of the corpora) on each of the test corpora.** Compute perplexity as follows:

$$PP = \left(\prod_{i}^{N} \frac{1}{P(W_i|W_{i-1},...,W_{i-n+1})}\right)^{\frac{1}{N}} = \exp \frac{1}{N} \sum_{i}^{N} - \log P(W_i|W_{i-1},...,W_{i-n+1})$$

[1] `http://qwone.com/~jason/20Newsgroups/`

where N is the total number of tokens in the test corpus and $P(W_i|W_{i-1}, ..., W_{i-n+1})$ is the n-gram probability of your model. Under the second definition above, perplexity is a function of the average (per-word) log probability: use this to avoid numerical errors.

6 Topic classification

In this part of the project, you will use language models to automatically predict the most likely topic that a new, unlabeled document should fall in.

Methodology suggestions. You are given plenty of labeled documents for each topic. To adjust your model for best performance (e.g. tuning smoothing parameters, or choosing between design options) you should divide the labeled data into a **training and development set** to evaluate the performance of your model. For this task, you will use the files/folders under the **classification_task** folder. Use the folder structure to determine the training and test sets as well as the class labels (e.g. all the articles in `atheism/train_docs` are training documents with category atheism). You should use only the training data for developing your model. **You should submit your predictions for the articles in the test data (the documents on `test_for_classification` folder) on Kaggle.** Keep in mind that the evaluation metric used on Kaggle is **accuracy**.

Hint. There are potentially many ways to use language models as predictors. The phrasing in the very first sentence in this section should give away one way that is probabilistically meaningful. You may explore other ways as well; regardless, **make sure to clearly explain your method, and why you chose it, in the report**. We don't recommend that you try fancy or advanced ideas without first implementing and evaluating the simple, straightforward one we intended. (Good life advice in general!)

6.1 Kaggle submission Format

Apart from your report, you will submit a **csv** file to Kaggle.
There should be one prediction on each line for each document inside the `test_for_classification` folder. The name of each file gives its `Id`. Each line should contain a document `Id` and the predicted category that your algorithm places the document under, separated by a comma. So your output file should look like:

```
Id,Prediction
file1.txt,0
file2.txt,2
...
```

where file1 and file2 are file names of the two files on the `test_for_classification` folder, and the prediction values are integers using the following encoding:

```
atheism: 0
autos: 1
```

```
computer_graphics: 2
medicine: 3
motorcycles: 4
religion: 5
space: 6
```

Do not forget to include the header line (`Id,Prediction`) at the beginning.

7 Context-aware spell checker

In this section, you are going to develop n-gram models to implement a context-aware spell checker. For this task, you will use files/folders under the **spell_checking_task** folder You are given a *confusion set* file that consists of words that are very similar to each other. In the `confusion_set` file, in each line, there are two words with similar spelling. For example, the first line includes the words 'want' and 'went'. For a given document with misspellings you should use your language model to determine which of these words ('want' or 'went') makes more sense in the given context when you encounter either of the two words.

For each topic, you are given training data that has files with the correct spelling and the corresponding files (with the same name) with misspellings. The files with the correct spelling are in the `train_docs` folder whereas the corresponding files with misspellings are in `train_modified_docs` folder. For each topic, you are also given the test documents with misspellings for which you have to produce the corresponding documents with correct spelling. Again, divide your training set into training and development sets to evaluate your method. Use your trained language model to correct the misspellings in a given file.

7.1 Submission Format

You will **not** submit the output for this task on Kaggle but you will submit it on **CMS**. Keeping the folder structure the same, output the corrected files for each file on the `test_modified_docs` folders on `test_docs` folders. The files you output on `test_docs` folders should have the same names as their corresponding files on `test_modified_docs` folders. On CMS, upload **only** `test_docs` folders (e.g. you should be uploading data folder with topic folders such as atheism, autos etc. Topic folders should include **only** `test_docs` folders).

8 Open-ended extension

For the final part of the project, **decide on at least one additional extension** to implement using the n-gram statistics, random sentence generator, and perplexity computation as tools. The idea is to identify some aspect of the language model or random sentence generator to improve or generalize, and then to implement an extension that will fix this issue or problem. Below, we provide some ideas. In addition, it is important to **evaluate your extension quantitatively and/or qualitatively** to determine whether or not the extension obtained the expected behavior.

8.1 Extension Ideas

Below are some ideas for extensions. Alternatively, you are free to come up with your own extension. Whatever you choose to do, **explain the extension clearly in your writeup**.

- Implement a trigram (or 4-gram, or a general n-gram) model. Measure the perplexity of both training and test corpora as you increase n. Interpret the results you observe regarding the changes in perplexity.
- Implement another smoothing method.
- Implement an interpolation method (e.g. Linear interpolation, Deleted interpolation, Katz's backoff).
- Develop and implement a method for better handling of unknown words.
- Employ the language model in the service of another NLP or speech application.
- Redo part (or parts) of the exercise at the sentence level, i.e. by modeling sentences out of their context instead of modeling the whole text (books). In this case you will need to define a way to measure the probability of a text that is larger than sentence, explain your assumptions regarding this. Explain the advantages / disadvantages of modeling text this way.
- An n-gram language model is just a function from n words to a real number (probability) that is implemented as a lookup table (alternatively, you can see it as a function of $n - 1$ words to a distribution over your vocabulary). Define this function some other way than a lookup table (e.g. many machine learning methods that compute a function of a sequence of words to a real number: you may want to look up *neural language models*). Describe a simple method to estimate that function from data. Implement this method.
- Redo part (or parts) of the exercise using Brown clusters[2] and collapsing the word types that belong to a cluster into a single word type. For random sentence generation, you will need to define a way to generate sentences, since you will need to generate words from clusters. You may try Brown clusters at different granularities and explain the advantages / disadvantages. You don't need to "learn" the Brown clusters themselves i.e. you don't need to run the Brown clustering method. Just use pretrained Brown clusters.

[2] `http://metaoptimize.com/projects/wordreprs/`

9 Grading rubric

Part 1

- Unigram and bigram probability table; random sentence generator (10%)

Part 2

- Smoothing, unknown word handling, and perplexity calculation (10%)
- Implementation of perplexity (10%)
- Your extension of choice (10%)
- Topic classification (10%)
- Context-aware spell checking (10%)
- Experiment design and methodology; report quality (35%)
- Kaggle submission (5%)

Note: You may make up some of the lost points in Part 1 if you submit a corrected version together with Part 2. (Maximum points you can make up this way is 7/10 for Part 1).

Performance concerns: While you shouldn't focus on optimization, nothing in this assignment should need to be slow. If you find your implementations taking really long amounts of time, something is probably wrong with your design. You may lose points for impractically slow, brute-force style solutions, where applicable.

10 What to submit

Part 1 (via CMS): Submit a zip file with your code. Also submit examples of the random sentences produced and the analysis of the generated sentences (in a pdf file).
Part 2 (via CMS): Check the submission format for each of the tasks as explained above. Submit your (1) report (a .pdf), (2) code, and (3) data folder produced in the 7th section on CMS as a .zip or .tar file. Also, submit (4) the csv file produced (for section 6) on Kaggle.

In order to be graded as a group, **you must form groups with your partner(s) on Kaggle**. You will have a limited number of Kaggle submissions per group.

11 The report

With the the Part 2 submission, you should submit a short document (no more than 6 pages) that describes your work. Your report should contain **a section for each of Sections 2–8 above** as well as **a short *workflow* section** that explains how you divided up the work for the project among group members. **Important:** the role of this document is to show us that you **understand** and

are able to communicate what you did, and how and why you did it. **Come see us if you're not sure how to answer either of these questions!**

You might think this means you must write a lot, or include lots of code. If you find yourself needing to write a lot to explain something, it may be a bad sign. Describe the general approach, the data structures used (where relevant). Use examples. Identify all design choices and justify them (e.g. how did you deal with unknown words? how did you smooth?) Wherever possible, **quantify your performance** with evaluation metrics (e.g. report classification performance on the development set you set aside). If you tried any extensions, perform measurable experiments to show whether or not your extension had the desired effect? If something doesn't work or performs surprisingly, try to look deeper and explain why.

Code. You may include snippets of your code in the report if you think it makes things clearer. Include **only** relevant portions of the code (such as a few lines from a function that accomplish the task that you are describing in a nearby paragraph, to help us understand your implementation). Including boilerplate code or tedious, unnecessary blocks (e.g. reading files) only makes your report cluttered and hard to follow, so avoid it.