Assignment 1

CMSC 473/673 — Introduction to Natural Language Processing

Due Saturday September 23rd, 2017, 11:59 PM

Item	Summary
Assigned	Wednesday August 30th, 2017
Due	Saturday September 23rd, 2017 (originally Wednesday September 20th)
Topic	Counting, Probability, and Language Modeling
Points	150

In this assignment you will step through some introductory NLP techniques.

You are to *complete* this assignment on your own: that is, the code and writeup you submit must be entirely your own. However, you may discuss the assignment at a high level with other students or on the discussion board. Note at the top of your assignment who you discussed this with or what resources you used (beyond course staff, any course materials, or public Piazza discussions).

The following table gives the overall point breakdown for this assignment.

Question	1	2	3	4	5	6
Points	15	15	20	30	20	50

This handout may be lengthy, but think of it as both a tutorial and assignment. I provide a lot of explanation and re-summarizing of course concepts.

However, because this assignment handout *is* lengthy, I am first providing a **task list**. This task list captures the questions, and details (without other explanatory text) the tasks you are to do and what your completed assignment should answer. Following the task list are the **full questions**. The full questions do *not* require you to answer additional questions.

The task list enumerates what you must do, but it does not necessarily specify *how*—that's where the full questions come in. Therefore, you **should still read and reference** the full questions.

What To Turn In Turn in a writeup in PDF format that answer the questions; turn in all requested code necessary to replicate your results. Be sure to include specific instructions on how to build (compile) your code. Answers to the following questions should be long-form. Provide any necessary analyses and discussion of your results.

How To Submit Submit the assignment using the `submit` utility on GL. The course id is `cs473_ferraro`. This assignment's id is `a1`.

Task List

1. (a) How many sentences are there in the training and development splits? On average, how many sentences are in each document? Briefly explain if and why you were (or were not) suprised at this average.
   (b) In the training split, how many different word types and tokens are there?
   (c) Examine some of the most common words, like the twenty, thirty, and fifty most common ones. Discuss what you notice about those common words. Hypothesize some ways that you could group together words, and what effect(s) this collapsing would have. You can argue for or against your collapsing method. You do not need to implement this collapsing method.
   (d) Examine some of the least common words, such as words that appear only once, ten, fifty and 100 times. Discuss your opinion on how standard and/or reasonable these words are (e.g., would you want to keep them around in a vocabulary, or would you want to spend the computational resources to deal with them?).
   (e) How many word tokens are there in the development set? How many of these words were not seen in the training data?

2. (a) From the training set, plot the number of words appearing n times (i.e., $|V_n|$) vs. n. Discuss what you observe, what (if any) transformations you had to use in order to better analyze the data, and the implications of your observations.
   (b) Let U_n be the set of out of vocabulary words appearing n times (in the development corpus). Plot $|U_n|$ vs. n and discuss what you observe.

3. Examine (experimentally) Zipf's law on the provided training set. Experiment with and report your findings from at least three different *configurations* (combinations of preprocessing). Include a description of your methodology, justification for your configuration choices, the results, and an analysis.

   Your analysis should include plots and be quantitative, and it should include a discussion of results from linear regression (ordinary least squares). What does the slope of linear regression tell you? What about the coefficient of determination (the R^2 value)?

4. Read and write a half page summary and review of Church and Hanks (1989). In addition to discussing the basic methodology and findings of this paper, identify findings you found interesting, surprising, or confusing. What is the overall takeaway (for you) from this paper?

5. This question is purely analytical (written)—it does not require coding.
   (a) Derive an expression for the bigram backoff weight $\alpha(x)$. Show the steps involved in the derivation.
   (b) Replace the unigram frequency $f(y)$ in (1) with a unigram backoff model (2), and derive an expression for the unigram, and if necessary bigram, constant backoff weights β and α that ensure all models form proper distributions.

6. Perform and write up well-controlled examinations of three classes of language models. Document internal progress and development by training on the training set and evaluating on the development set. This internal development **must** include some amount of tuning of hyperparameters. When internal development is done, you are to pick the best configuration, including value of N and parameter values, for each model class. Once that is done, evaluate these best-in-class models on the testing set.

Present the results, both internal and final, in readable formats, e.g., in tables or plots. Be sure to describe, and justify, any design decisions or assumptions you made along the way. Discuss and document any tests or verifications you performed to ensure your models form proper probability distributions. Identify the best language model overall.

Turn in all code, including what is necessary for producing the plots. Also turn in the serialized version of the best performing language model; clearly indicate in the writeup what model and model configurations resulted in the best test set performance.

Full Questions

1. (**15 points**) In this question, you'll be doing some basic counting of words—the "Hello, World" of NLP.

   In the GL directory
   `/afs/umbc.edu/users/f/e/ferraro/pub/473-f17/a1/data/txt/`
   you will find 10,000 random English Wikipedia articles. They have been split into three different sections, or *splits*: 5,000 training documents (`train.5k.txt.gz`), 2,000 development documents (`dev.2k.txt.gz`) and 3,000 heldout (testing) documents (`test.3k.txt.gz`). **There have been some permission issues.** While they get sorted out, I have mirrored the three files online, at `https://www.csee.umbc.edu/courses/undergraduate/473/content/materials/a1/data/txt/train.5k.txt.gz`, `https://www.csee.umbc.edu/courses/undergraduate/473/content/materials/a1/data/txt/dev.2k.txt.gz`, and `https://www.csee.umbc.edu/courses/undergraduate/473/content/materials/a1/data/txt/test.3k.txt.gz`.

   These files are text files that have undergone gzip compression to save space. Each line is a different sentence, and "words" are space-separated. Words have sequences of non-whitespace unicode characters.

   It's a good idea to look at your (training) data. From the command line, you can get a sense of what you're dealing with by randomly sampling some number, say 10, of sentences. First change the working directory to where the data files are

   ```
   $ cd /afs/umbc.edu/users/f/e/ferraro/pub/473-f17/a1/data/txt
   ```

   then decompress the training file in place and pipe the decompressed output to the `shuf` command.[1]

   ```
   $ gzip -d -c train.5k.txt.gz | shuf -n 10
   years3 :| clubs3 = Faversham Town F.C. | Faversham Town | caps3 = | goals3 =
   As with most role-playing games , these four attributes have a direct impact on the way the player 's character evolves .
   The film got a great opening and the film went on to become a hit .
   nationality : Spanish
   Approximately 2 million mines were laid in various areas of Croatia during the war .
   According to biographer Vladimir Lakshin , Rogozhin in Dostoyevsky 's The Idiot bore similarities to Krasnov .
   He later attended rookie mini-camp with the Chicago Bears and worked out for the Detroit Lions , but was n't offered a contract by either team .
   first_aired : September 17 , 1991
   ```

[1] The `-d` flag to `gzip` performs the decompression while the `-c` flag writes the output to `stdout` (the terminal) rather than to a file on disk. Without the `-c` flag `gzip` would try to replace the compressed file with an uncompressed one (but it would fail since you do not have the correct permissions); output would not be written to the terminal. As with many Linux commands, you can combine the short flag options together: the command `gzip -d -c` is the same as `gzip -c -d`, which is the same as `gzip -cd` and `gzip -dc`.

The `shuf` command shuffles the input; in this case, the input is implicitly given as reading from `stdin`. The `-n` flag indicates how many lines to sample.

Want to learn more about text processing with Linux? Ken Church's Unix for Poets is a great start.

```
The work in his new book is as strong as ever , and we 're excited
   to be publishing him '' .
birth_place : Roscrea , County Tipperary
```

Well, that looks a bit strange, doesn't it? And no, you're not looking at formatting errors. Now, when you run the above command, you'll get different output, but it should display roughly the same characteristics. You have punctuation separated from words (`evolves.` → `evolves .`), single word contractions (`wasn't` → `was n't`) and possessives split (`player's` → `player 's`), and some weird looking sentence fragments and words.

Unless specified otherwise, use the **training** split.

(a) How many sentences are there in the training and development splits? On average, how many sentences are in each document? Briefly explain if and why you were (or were not) suprised at this average.

There's no right-or-wrong answer for if you were surprised: try to think about the kinds of documents you read, and characteristics of those documents.

You can use any method you want to compute the number and averages (e.g., Linux commands, or a small Python or Java program). Turn in what you wrote, or provide the command in your writeup with a brief explanation of the steps involved.

The text you see above is *tokenized text*. In particular, it is text that has been tokenized, or split into individual words, according to the Penn Treebank (PTB) specification.[2] By splitting contractions and possessives, PTB tokenization provides some very basic morphosyntactic analysis of English.[3] PTB has a number of rules for punctuation, though the weirdest looking may be terms like `-LRB-`. The following table gives the correspondence among (R)ound, (S)quare, and (C)urly braces:

(	)	[	]	{	}
-LRB-	-RRB-	-LSB-	-RSB-	-LCB-	-RCB-

The first letter represents an opening (L—left) or closing (R—right) brace. The middle letter represents the shape. These mappings are necessary for some NLP tasks (in particular, how they were originally defined).

To dive into this some more, recall the token-type distinction. The individual instances you observe are tokens, where each token is drawn from a set of types. Using a programming analogy, we can say that word types are like classes while word tokens are like instances of that class.

In the above files, individual word tokens are white-space separated character spans. For example, in the third sentence shown above, and repeated here,

```
The film got a great opening and the film went on to become a hit .
```

there are 16 tokens instantiated from 14 types (the separator character does not count):

[2] The original, `ftp://ftp.cis.upenn.edu/pub/treebank/public_html/tokenization.html`, is actually fairly simple to implement. The actual tokenization used to produce the text for this assignment is a slight modification and generalization of the original.

[3] Morphosyntax studies the combination of morphology (the study of word forms) and syntax (the study of sentences and word order).

(b) In the training split, how many different word types and tokens are there? Do not perform any processing on the lists that modify the words. Turn in the code for this, or if done on the command line, describe how.

(c) Observe some of the most common words, like the twenty, thirty, and fifty most common ones. Discuss what you notice about those common words. Each word should be alphanumerically (lexically) distinct. But should all these items actually be considered distinct? What are some ways that we could group together words? Hypothesize some effects your collapsing would have. You can argue for or against your collapsing method.

As before, there is not a right or wrong answer here. Some collapsing methods may be more appropriate than others, but the question is to think about these methods and what effect they may have.

Now, there are *simpler* answers. In particular, some collapsing methods can be accomplished with simple calls to standard string processing functions. Others could be accomplished with some more advanced string processing, like regular expressions. Others still could use more advanced NLP methods (that we'll cover and you'll implement later on in the semester).

(d) Examine some of the least common words. Do words that appear only once look like "standard" words? What about words that appear ten times? Fifty times? What about 100 times?

Regardless of whether these words were standard, are they "reasonable?" Are they items that would want to be able to talk about as distinct items? From a computational point of view, do you want to spend the computational resources to deal with them?

Argue for or against these items' reasonableness. If you find them unreasonable, propose a solution. You do not need to implement it.

(e) Now it's time to look at the development split. How many word tokens are there? How many of these words were not seen in the training data? We call these *out of vocabulary* (OOV) words.

Again, do not perform any processing on the lists that modify the words. Turn in the code for this, or if done on the command line, describe how.

2. (**15 points**) Let V_n be the set of word (types) that appear n times in the (training) corpus. For example, V_1 are those words that appear just once, V_2 are those words that appear twice, and V_{150320} are those words that appear 150320 times.

(a) From the training set, plot the number of words appearing n times (i.e., $|V_n|$) vs. n. Discuss what you observe, what (if any) transformations you had to use in order to better analyze the data, and the implications of your observations.

(b) Let U_n be the set of out of vocabulary words appearing n times (in the development corpus). Plot $|U_n|$ vs. n and discuss what you observe.

For both parts of this question, turn in both the plot (either as a PDF, PNG or JPG) and any code needed to reproduce the data and plot.

3. (**20 points**) It's common to say that language follows a power law distribution. Alternatively, you may hear that 'language is Zipfian.' In this question, you'll explore what that means.

Imagine counting up words and performing a descending sort of the words according to how many times each appeared. That is, words that appear more often appear before words that appear less often—in such a setting, the more common words are said to have a lower rank. The basic Zipf

estimate states that the frequency $f(y)$ of a word y is inversely proportional to how common it is (its rank r),

$$f(y) = \frac{C}{r(y)},$$

where C is some constant. Taking logarithms results in a proposed linear relationship between the log frequency (count) and the log rank.

How well does Zipf's law hold? First, you should compute $f(y)$ and $r(y)$ for each word (type). But how do you define words? Should you lower case words? Should you examine $f(y)$ vs. $r(y)$? Or maybe should you examine the log of each? The low count words for larger datasets will be different than the low count words for smaller datasets: if Zipf's law holds for larger (smaller) datasets, does it hold for the other?

These choices—in particular whether or not to lower case words (or if you apply any other kinds of transformations to the words), and the amount of data to use—are yours. Each joint choice forms a configuration. Below are example configurations:

word transformation	**amount of data used**
none	100%
none	10% sampled uniformly at random
none	1% sampled uniformly at random
lower case	100% sampled uniformly at random
lower case	50% sampled uniformly at random
lower case	50% sampled with some bias

Experiment with and report your findings from at least three different configurations. You may consider different ways of defining configurations; these configurations can include any word transformations you defined above. Just be sure to describe what you did. Your analysis should include plots and be quantitative, and it should include a discussion of results from linear regression (ordinary least squares), which fits a line to provided independent and dependent variables.[4] What does the slope of linear regression tell you? What about the coefficient of determination (the R^2 value)? In your analysis, include a description of your methodology, justification for your configuration choices, the results, and an analysis of the results. Turn in all code needed to reproduce your analysis.[5]

4. (**30 points**) Read and write a half page summary[6] and review of Church and Hanks (1989). It is available on GL at

   `/afs/umbc.edu/users/f/e/ferraro/pub/473-f17/a1/church-hanks-1989.pdf`

   In addition to discussing the basic methodology and findings of this paper, identify findings you found interesting, surprising, or confusing. What is the overall takeaway (for you) from this paper?

   The full Bibtex citation is

```
@InProceedings{church-hanks:1989:ACL,
  author    = {Church, Kenneth Ward and Hanks, Patrick},
```

[4] In R, linear regression is available through the `lm` function, from the basic `stats` package (typically preloaded). Linear regression is available in Python through a number of methods, including from `scipy` under `scipy.stats.linregress`.

[5] Though we'll have your code, your methodology description should be thorough enough for us to reimplement what you did. One area to be careful of: if you subsample the amount of training data, how do you do it?

[6] Single spaced, regular font, and one column is fine.

```
  title     = {Word Association Norms, Mutual Information,
               and Lexicography},
  booktitle = {Proceedings of the 27th Annual Meeting of the
               Association for Computational Linguistics},
  month     = {June},
  year      = {1989},
  address   = {Vancouver, British Columbia, Canada},
  publisher = {Association for Computational Linguistics},
  pages     = {76--83},
  url       = {http://www.aclweb.org/anthology/P89-1010},
  doi       = {10.3115/981623.981633}
}
```

5. (**20 points**) This question analyzes smoothed bigram language models. You will implement a generalization of this model in the next question; therefore while this question requires no coding, you are welcome to verify what you arrive at computationally.

   In class we discussed how to create smoothed bigram language models $p(y|x)$. One way is to interpolate other language models. For example, we can define the interpolation model

   $$p_I(y|x) = \lambda f(y|x) + (1 - \lambda) f(y),$$

   where λ is set to optimize **development** corpora, and $f(\cdot|\cdot)$ and $f(\cdot)$ are the bigram and unigram relative frequencies, i.e., bigram and unigram maximum likelihood estimates,

   $$\begin{aligned} f(y|x) &\propto c(x, y) & f(y) &\propto c(y) \\ &= \frac{c(x, y)}{\sum_v c(x, v)} & &= \frac{c(y)}{\sum_v c(v)}. \end{aligned}$$

   The counts $c(\cdot, \cdot)$ and $c(\cdot)$ are obtained from **training** corpora.

   An alternative is to use **backoff** models. Whereas interpolation models average different probabilities, backoff models partition the items they model into different classes—they stratify the objects they model. Each class can then be modeled differently. In the remainder of this question, you will examine the backoff bigram model

   $$p_B(y|x) = \begin{cases} \dfrac{c(x, y) - \delta_2}{c(x)} & \text{if } c(x, y) > 0, \\ \alpha(x) f(y) & \text{otherwise.} \end{cases} \tag{1}$$

   Here δ_2 is a positive constant *discount factor*; δ_2 is less than 1 ($0 < \delta_2 < 1$). The multiplier $\alpha(x)$ is constant for each x; it is called the backoff weight (for x). Note that it is a function and that for different histories x_1 and x_2 it may give different values, i.e., $\alpha(x_1) \neq \alpha(x_2)$. This backoff weight is defined so as to make $p_B(\cdot|x)$ a proper probability distribution.

   (a) Derive an expression for $\alpha(x)$. Your expression should be in terms of the discount factor δ_2, the counts ($c(\cdot, \cdot)$, $c(\cdot)$), and the unigram frequency (probability) $f(y)$. Show the steps involved in the derivation.

(b) What if y is never observed? This happens when both $c(x, y) = 0$ and $f(y) = 0$. Let's iterate the backoff procedure and replace $f(y)$ with a unigram backoff model,

$$q_B(y) = \begin{cases} \frac{c(y) - \delta_1}{\sum_v c(v)} & \text{if } c(y) > 0, \\ \frac{\beta}{V} & \text{otherwise,} \end{cases} \tag{2}$$

where V is the size of the vocabulary (the number of items over which the p_B and q_B are defined). Like δ_2, δ_1 is a constant discount factor, and β is a constant backoff weight. Redefine p_B in terms of q_B,

$$p_B(y|x) = \begin{cases} \frac{c(x, y) - \delta_2}{c(x)} & \text{if } c(x, y) > 0, \\ \alpha(x) q_B(y) & \text{otherwise.} \end{cases} \tag{3}$$

Derive an expression for β so that both p_B and q_B form proper probability distributions. If necessary, also re-derive an expression for $\alpha(x)$.

6. (**50 points**) You're going to gain experience implementing and evaluating a variety of language models: this question is a combination of engineering and art.

Your Task Your task is to perform well-controlled examinations of the following models (described below). Document internal progress and development by training on the training set and evaluating on the development set. This internal development **must** include some amount of tuning of hyperparameters. When internal development is done, you are to pick the best configuration, including value of N and parameter values, for each model class. Once that is done, evaluate these best-in-class models on the testing set. Present the results, both internal and final, in readable formats, e.g., in tables or plots. Be sure to describe, and justify, any design decisions or assumptions you made along the way. Discuss and document any tests or verifications you performed to ensure your models form proper probability distributions. Identify the best language model overall.

Turn in all code, including what is necessary for producing the plots. Also turn in the serialized version of the best performing language model; clearly indicate in the writeup what model and model configurations resulted in the best test set performance.

How You Will Be Graded Your grade on this problem will be from proper documentation of your code, experiments, and reporting of results on both the development and test portions. Your code will also be tested on some new data—two completely novel test sets that you will not have access to. One of the test sets will be drawn from the same Wikipedia source as you've been using; the other will not. As long as your model is non-trivial and returns a finite perplexity, your performance, i.e., the actual perplexity value, on these two novel datasets will **not** affect your *base* grade.

However, as a reward the top three performers on these surprise data will receive **bonus points**: 8 bonus points for the top performer, 5 points for the second top performer, and 3 points for the third top performer. These points are awarded independently for each of the surprise datasets, the person receiving the points (one person could receive 16 bonus points), and 473 vs. 673 (there will be up to six undergraduate and up to six graduate awardees).

Basic API Your code must be callable through two bash scripts. The **first**, `ngram_lm_train.bash`, trains a specified model given training and development sets. It then serializes the learned model. This script will be called as

```
$ bash ngram_lm_train.bash model-name N \
```

```
path/to/train.txt.gz path/to/tune.txt.gz \
path/to/serialize/model
```

where `model-name` is one of the three recognized names {`laplace`, `interpolate`, `backoff`} (described more below), `N` is an integer specifying the Markov order (plus 1) of the model to train. That is, when $N = 1$ you should train unigram models, and when $N = 3$, train trigram models.

The first path argument provides the location of the (gzipped) text that will provide the basic counts, and the second path argument provides the location of the (gzipped) text you will use to tune any parameters of the specified model. Finally, after doing all the work of counting and optimizing any model parameters, you need to save the model to disk; the third path argument provides this location. The serialized model must contain all of the necessary information to reconstruct and use the model later on, without having to specify the model name or Markov order.

The **second** script evaluates your saved model (the first argument) on heldout data (the second argument):

```
$ bash ngram_lm_eval.bash path/to/serialized/model \
    path/to/eval.txt.gz
```

The evaluation is perplexity.

Models Implement the following parametrizable models. Your code must be able to handle $N \in \{0, 1, 2, 3\}$.

(a) (model name: **laplace**) N-gram models with Laplace smoothing ($\gamma > 0$),

$$p_\gamma^{(n)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) \propto c(x_{i-n+1}, \ldots, x_i) + \gamma.$$

(b) (model name: **interpolate**) N-gram models with interpolation of n-gram frequencies with lower-order (m)-gram frequencies, for $0 \leq m < N$. This can be done either directly, as the interpolation of $N + 1$ items, or recursively, as the interpolation of an n-gram frequency and an (interpolated) $n - 1$ probability.

The first case (the interpolation of $N + 1$ frequencies) is written as

$$\begin{aligned} p^{(n)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) &= \sum_{m=0}^{n} \lambda_{n,n-m} f^{(n-m)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) \\ &= \lambda_{n,n} f^{(n)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) + \lambda_{n,n-1} f^{(n-1)}(x_i|x_{i-n+2}, \ldots, x_{i-1}) + \\ &\quad \lambda_{n,n-2} f^{(n-2)}(x_i|x_{i-n+3}, \ldots, x_{i-1}) + \ldots + \lambda_{n,0} \frac{1}{V}, \end{aligned}$$

where $f^{(0)} = \frac{1}{V}$ is the uniform distribution. So the trigram model interpolates trigram, bigram, and unigram frequencies with a uniform estimate. Although there are $N + 1$ interpolation parameters written, they must all sum to 1: $\sum_{m=0}^{n} \lambda_{n,n-m} = 1$. Using this information, you can rewrite any one of them in terms of the others, e.g,, $\lambda_{n,0} = 1 - \sum_{m=0}^{n-1} \lambda_{n,n-m}$.

The second case (recursive definition) is written as

$$p^{(n)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) = \lambda_n f^{(n)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) + (1-\lambda_n) p^{(n-1)}(x_i|x_{i-n+2}, \ldots, x_{i-1}).$$

In both cases, we define the base case $n = 0$ to be the constant unigram model,

$$f^{(0)}(\cdot) = \frac{1}{V}.$$

Each interpolation model $p^{(n)}$ is the interpolation of two frequencies.

(c) (model name: **backoff**) N-gram models with recursive backoff,

$$p^{(n)}_{B,\delta_n}(x_i|x_{i-n+1},\ldots,x_{i-1}) = \begin{cases} \dfrac{c(x_{i-n+1},\ldots,x_i)-\delta_n}{c(x_{i-n+1},\ldots,x_{i-1})} & \text{if } c(x_{i-n+1},\ldots,x_i) > \epsilon_n, \\ \alpha_n(x)p^{(n-1)}_{B,\delta_{n-1}}(x_{i-1}|x_{i-n+1},\ldots,x_{i-2}) & \text{otherwise.} \end{cases}$$

You may decide how to define the base case $n = 0$.

Note the generalization in the first case: rather than set the count threshold to any sequence occurring zero times, what if you set it to be a *positive* integer ϵ_n?

Evaluation You will evaluate a language model p using perplexity:

$$\text{ppl}(p) = \exp\left(\frac{-1}{M}\sum_{i=1}^{M}\log p(w_i|h_i)\right). \tag{4}$$

The logarithm here is assumed to be the natural logarithm. The sum iterates over history-observation instances h_i, w_i; depending on the order of the model under consideration, the history h_i may actually be a sequence of words. There are M of these instances. This is the "token" view, i.e., we process each observed word (given its history) in turn.

You can alternatively compute perplexity after counting up the unique history-observation pairs:

$$\text{ppl}(p) = \exp\left(\frac{-1}{M}\sum_{x,y}c(x,y)\log p(y|x)\right). \tag{5}$$

Here there are W unique history observation pairs x, y. This is the "type" view, i.e., we first count up the number of different history-observation pairs, and then process that list. In general, the "type" view will involve fewer summands than the "token" view.

Note that in both cases, the history is actually a *sequence* of the $N-1$ previous words. This value N is provided via the language model p.

Tuning the Models The three classes of models you will implement have some tunable parameters. You will probably want to tune them using the development portion; however, you must **not** tune them using the test portion.[7]

You need to include a special, **unique** OOV symbol in the vocabulary that represents *all* out-of-vocabulary words. All evaluation sets need to remapped (internally, or on the fly) to the training vocabulary; words seen in evaluation but not seen in training must be mapped to this OOV symbol. But will you ever actually see OOV in training? A common technique is to identify "rare" words in training and map them to the OOV symbol. However, what is "rare?" Sometimes it's any words that appear just once. Other times it's words that appear fewer than five times. And other times it's words that appear fewer than 100 times. This rare words threshold represents another tunable parameter.

There are a couple of ways to tune the models. Let $\theta_1,\ldots,\theta_T$ be tunable parameters. The first way to tune the values θ_t is through a grid search. Grid search selects a value for each θ_t, updates any necessary model parameters, and evaluates the model (on the development set), reporting both the end evaluation score and the values of θ_t. The final values for θ_t are those that resulted in the best evaluation performance.

[7] What would you use instead of the development portion? Sometimes people will further split the training portion for initial development. This allows different parameters to be tuned with somewhat different statistics, hopefully limiting overfitting.

Grid search often uses fixed values. These values are generally arbitrary, but they tend to be "nice" numbers like powers of 10 ($\{0.1, 1, 10, \ldots\}$) or powers of 2.

Grid search is easy to set up, but it can be difficult to do this efficiently or effectively (why are the values you chose good ones at all?). An alternative is to directly optimize the evaluation objective, with respect to the tunable parameters. The most common approach is through a gradient-based optimization procedure called gradient descent (or ascent).

Gradient descent minimizes a differentiable objective $J(\Theta)$ with respect to $\Theta = \{\theta_t\}$. Gradient descent proceeds iteratively: given values of the variables at time s, the values at time $s+1$ are found by translating the current values by the scaled gradient of J evaluated at $\Theta^{(s)}$:

$$\Theta^{(s+1)} = \Theta^{(s)} - \rho_s \nabla_\Theta J(\Theta)|_{\Theta^{(s)}}. \tag{6}$$

The scaling factor is ρ_s: setting it is an entire area of active research. However, an effective[8] learning rate is

$$\rho_s = \frac{\rho_0}{1 + \rho_0 \frac{\kappa}{M} s}, \tag{7}$$

where ρ_0 is an initial learning rate, and both ρ_0 and κ are some (you guessed it) tunable constants. Setting $\kappa = 1$ is reasonable, though.

Note that if you have a function $g(x) = \exp(Ch(x))$, where C is a (positive) constant, then a value $x^\star$ minimizing h also minimizes g. Although the final evaluation for this problem is perplexity, the objective J can actually just be part of the argument to the exp function:

$$J(\Theta) = -\sum_{x,y} c(x, y) \log p_\Theta(y|x). \tag{8}$$

This is the *negative log-likelihood*.

Now, what can you tune? Integer-valued parameters, like ϵ_n in the **backoff** models are difficult to optimize (the objective ends up being non-differentiable and piecewise constant). These parameters are best left to grid search.

So what about the others? Although all of the other parameters are real-valued, they are *constrained* real values. Specifically, they are all positive (> 0), and the interpolation weights λ_n have the additional constraint of being less than 1. Constrained optimization is generally harder than unconstrained optimization. Let's consider two ways to do constrained optimization.

The first is called projected gradient descent (PGD). With PGD, you make the update as given by (6). However, you then define a *projection map* Π that projects $\Theta^{(s+1)}$ into the constrained space. The projected point $\Pi(\Theta^{(s+1)})$ is the point that is closest (under the standard Euclidean distance) to $\Theta^{(s+1)}$ that satisfies all of the constraints. So if $\Theta^{(s+1)}$ already satisfies the constraints, then $\Pi(\Theta^{(s+1)}) = \Theta^{(s+1)}$. For the positivity constraints, you can just set any negative components to a small, positive number. In this case, the same approach can be applied to the unit (or simplex) constraints, $0 < \lambda < 1$.

A second approach lies in reparametrizing the values. Consider the count adjustments $\gamma_n > 0$ for Laplace smoothing. You could define $\gamma_n = \exp(\hat{\gamma_n})$, where $\hat{\gamma_n} \in \mathbb{R}$ is an unconstrained real number. Now you optimize J with respect to $\hat{\gamma_n}$, rather than γ_n. While this can be effective, you must apply the chain rule in order to properly compute the gradient, $\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f(g(x))}{\partial g(x)} \frac{g(x)}{x}$. A function like the sigmoid function, $\sigma(x) = \frac{1}{1+\exp(-x)}$, allows a reparametrization for the interpolation weights.

[8] See Leon Bottou's "Stochastic Gradient Descent Tricks" for more.

Though it can be simple to write your own gradient descent optimizer, you are allowed to use an external optimization library to perform gradient-based tuning. This includes using neural net frameworks to help with the gradient computation. However, regardless of what (if any) library you use, if you apply gradient-based tuning you **must** derive and document the gradient in your writeup.

Serializing the Model The serialized model must contain all of the relevant information for loading and using the model. In particular, it must know N, and contain all relevant counts, values of tunable parameters, and the known vocabulary.

Implementation and Analysis Considerations There are a number of implementation issues to consider. Let's look at a few of them.

The first is how you store the probabilities. You are implementing recursive N-gram models over a large vocabulary—let's assume the vocabulary has 50,000 words. If you represent all possible probabilities, then even with single precision floating point values (32 bit), while a unigram model will use 200 kilobytes of memory, a bigram model will use 10 gigabytes! However, many of these will be 0 (or some trace smoothed amount). You should store only the information and counts you actually observe.

The second is to properly computing and handling the vocabulary. Recall the vocabulary must include a special, **unique** OOV symbol that represents *all* out-of-vocabulary words, and you have control over what words map (on the fly) to this OOV symbol. However, you need to be aware of the conclusions you draw when comparing perplexities from models that are defined over different vocabularies.

Third, you have incredible freedom in how you structure your code. Your code does not have to be pristine. However, we do need to be able to follow what you write. Ensure your code is appropriately commented and described in your writeup.

Try to make the coding easy on yourself. The recursive definitions permit rather simple methods for constructing higher-order N-gram models.

Fourth, you have incredible freedom over how you serialize the language models. Feel free to use your programming language's default way to serialize data. However, beware of versioning issues like `java.io.Serializable` ones found in Java.

Finally, the two bash scripts you write can, under the hood, call the same language model program. You do not have to write two totally separate programs; having code shared between your two programs is fine.