Assignment 3

CMSC 473/673 — Introduction to Natural Language Processing

Due Friday October 11th, 2019, 11:59 PM

<table>
<thead>
<tr>
<th>Item</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assigned</td>
<td>Monday September 23rd, 2019</td>
</tr>
<tr>
<td>Due</td>
<td>Friday October 11th, 2019</td>
</tr>
<tr>
<td>Topic</td>
<td>Probabilistic Language Modeling and Noisy Channel Classification</td>
</tr>
<tr>
<td>Points</td>
<td>135</td>
</tr>
</tbody>
</table>

In this assignment you will implement and evaluate probabilistic language models. You will use them to build noisy channel classifiers.

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.

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>20</td>
<td>10</td>
<td>25</td>
<td>10</td>
<td>20</td>
<td>20</td>
<td>30</td>
</tr>
</tbody>
</table>

You have incredible freedom in how you structure your code. Your code does not have to be pristine. However, we should 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.

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 on the submission site:

[https://www.csee.umbc.edu/courses/undergraduate/473/f19/submit](https://www.csee.umbc.edu/courses/undergraduate/473/f19/submit)

Be sure to select “Assignment 3.”
1. (20 points) This question analyzes smoothed bigram language models. You will implement a generalization of this model in a later question; therefore while this question requires no coding, you are welcome to verify what you arrive at computationally. Please turn in any code you use and discuss how you used it to verify your solution. Not turning in code will not adversely impact your grade (though it may provide an avenue for partial credit if a derivation is wrong).

Let \( f(\cdot|\cdot) \) and \( f(\cdot) \) be the bigram and unigram relative frequencies, i.e., bigram and unigram maximum likelihood estimates,

\[
\begin{align*}
    f(y|x) &\propto c(x,y) = \frac{c(x,y)}{\sum_v c(x,v)} , \\
    f(y) &\propto c(y) = \frac{c(y)}{\sum_v c(v)} .
\end{align*}
\]

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

**Backoff** models partition the items they model into different classes—they stratify the objects they model. Each class can then be modeled differently. This question examines the backoff bigram model

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

Here \( \delta_2 \) is a positive constant discount factor; \( \delta_2 \) 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 \( \delta_2 \), the counts \( c(\cdot, \cdot) \), and \( c(\cdot) \), and the unigram frequency (probability) \( f(y) \). Show the steps involved in the derivation. **Hint:** use what you know must be true of distributions and solve for \( \alpha(x) \).

(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,

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

where \( V \) is the size of the vocabulary (the number of items over which the \( p_B \) and \( q_B \) are defined). Like \( \delta_2 \), \( \delta_1 \) is a constant discount factor, and \( \beta \) 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 \( \beta \) so that both \( p_B \) and \( q_B \) form proper probability distributions. If necessary, also re-derive an expression for \( \alpha(x) \).
2. (10 points) Each of the following scenarios describes the result of some (made up) classifier: there is a list of the correct labels, and the corresponding predictions. For each of the following situations, compute accuracy, recall, precision, and $F_1$ score (at the macro level). You must show work, or some intermediate steps, to receive full credit on this problem.


(c) A multiclass classification result, where the correct labels are $[T, F, U, F, F, F, U, T]$ and the predicted labels are $[F, T, U, F, F, F, F, T]$. 


3. (25 points) This question is the first in a series that asks you to implement and evaluate a number of different $N$-gram language models that apply at the sentence level. These implementations and scripts developed here are needed for subsequent questions. All of the individual training or evaluation calls—for this and future questions in this assignment—should run at most within a couple of minutes (with the simpler ones like mle and laplace finishing in around 10-20 seconds).

To receive full credit on this problem, you must

- implement (and turn in your code for) the four models described below (see Models),
- properly implement the wrapper script (see Wrapper Script),
- briefly discuss, in your writeup, the code layout/design choices made,
- discuss how you handled the OOV, BOS, and EOS symbols (including any hyperparameters that could affect their use and the proper inclusion or exclusion from computing the probability distributions),
- decide on a Universal Dependencies dataset to use (see Dataset),
- give a detailed discussion in your writeup of how you verified your implementations are correct (this can be concise),
- show that you can train a model and compute dev perplexity results by completing the following table (see Perplexity). Your perplexity results do not have to be optimized.

<table>
<thead>
<tr>
<th>Basic Model Type</th>
<th>N-gram</th>
<th>Hyperparameters</th>
<th>Dev perplexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>mle</td>
<td>1</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td>laplace</td>
<td>1</td>
<td>$\gamma = ???$, ???</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$\gamma = ???$, ???</td>
<td>???</td>
</tr>
</tbody>
</table>

Models Implement the following four language models: (i) a unigram maximum likelihood (unsmoothed) model, (ii) a bigram maximum likelihood (unsmoothed) model, (iii) a unigram Laplace model, (iv) a bigram Laplace model.

Your code must be able to handle $N \in \{1, 2\}$, i.e., unigram and bigram models.
(a) (model name: mle) $N$-gram models with no smoothing,
\[ p^{(n)}(x_i|\ldots, x_{i-n+1}) \propto c(x_i-n+1, \ldots, x_i). \quad (4) \]

(b) (model name: laplace) $N$-gram models with Laplace smoothing, with hyperparameter $\gamma > 0$,
\[ p^{(n)}_{\gamma}(x_i|\ldots, x_{i-n+1}) \propto c(x_i-n+1, \ldots, x_i) + \gamma. \quad (5) \]

*Hint: can one model be implemented as an instance of the other? If so, make use of that fact.*

**Wrapper Script** These must be callable through a bash script `ngram_lm_train.bash`, which trains a specified model given a training set. It then computes the perplexity of that learned model on a development set. Finally it serializes the learned model to disk. This script will be called as

```
$ bash ngram_lm_train.bash model-name N \\
    path/to/train.conllu path/to/tune.conllu \\
    path/to/serialize/model [...hp-args...]
```

where `model-name` is one of the three recognized names `{mle, laplace}` and $N$ is an integer specifying the $N$-gram size to use. When $N = 1$ you should train unigram models, and when $N = 2$, train bigram models.

The third argument provides the location of the CoNLL-U formatted file that will provide the training text, and the fourth argument provides the location of the 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 fifth 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, $N$, or hyperparameters. Most languages provide a native ability to do this easily; in Python, you have `pickle`, and in Java you have `java.io.Serializable`.

Use `[..., hp-args...]` to pass in hyperparameters for the model in question. While these should override reasonable default values, explicitly listing `[..., hp-args...]` must not be necessary to get your code to run properly! The exact format of this argument is up to you and it depends, among other considerations, on the programming language you use and the values you consider to be hyperparameters. While a value like $\gamma$ for Laplace smoothing is an obvious hyperparameter, you could consider other values, like the frequency cutoff for OOV, as one too. (Beyond the “obvious” hyperparameters for each model, don’t get caught up on what “should” and “shouldn’t” be one. Discuss and justify in your writeup what you do/do not consider to be these non-obvious hyperparameters.)

**Dataset** You must experiment with at least one language, of your choice, from the Universal Dependency dataset. You may experiment with multiple languages, but you are not required to do so (and not doing so will not penalize you). Regardless of the language(s), **model each sentence separately**.

**Perplexity** You will evaluate each language model $p$ using perplexity,
\[ \text{ppl}(p) = \exp \left( \frac{-1}{M} \sum_{i=1}^{M} \log p(w_i|h_i) \right). \quad (6) \]

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:

\[
ppl(p) = \exp \left( -\frac{1}{M} \sum_{x,y} c_{\text{eval}}(x,y) \log p(y|x) \right). \tag{7}
\]

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 in the data we’re evaluating (represented with \( c_{\text{eval}} \)), and then process that list. Note that this \( c_{\text{eval}} \) is different from the counts you obtained in training (unless you’re evaluating on your training set). 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 \( p \).

4. (10 points) Now create a second wrapper script that can evaluate your saved model (the first argument) on new heldout data (the second argument):

\[
$ \text{bash ngram_lm_eval.bash path/to/serialized/model } \text{path/to/eval.conllu}$
\]

This script must compute and print out the perplexity on the heldout data. To receive full credit for this question:

- discuss how you handled the deserialization of the model,
- discuss (and turn in) any new code you wrote for this (building/extend the code you wrote for Q3 is completely okay and encouraged!),
- show that your script/code can compute perplexity. Use your new wrapper script to recompute dev perplexity on the same dataset you used in Q3; add these numbers to your previous table (or replicate and extend your previous table):

<table>
<thead>
<tr>
<th>Basic Model Type</th>
<th>N-gram</th>
<th>Hyperparameters</th>
<th>Dev ppl.</th>
<th>Dev ppl. (ngram_lm_eval)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mle</td>
<td>1</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>laplace</td>
<td>1</td>
<td>( \gamma = ??, ?? )</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>( \gamma = ??, ?? )</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

5. (20 points) Extend your scripts and code to handle backoff N-gram models for \( N \in \{1,2\} \):

- (model name: backoff) N-gram models with recursive backoff, \( P_{B,\delta_n}^{(n)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) = \)

\[
\begin{cases} 
  c(x_{i-n+1}, \ldots, x_i) - \delta_n 
  & \text{if } c(x_{i-n+1}, \ldots, x_i) > \epsilon_n, \\
  \frac{c(x_{i-n+1}, \ldots, x_{i-1})}{c(x_{i-n+1}, \ldots, x_{i-1})} 
  & \text{otherwise}. 
\end{cases} \tag{8}
\]

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 \( \epsilon_n \)?

To receive full credit on this problem, you must
• implement (and turn in your code for) the backoff models,
• briefly discuss, in your writeup, the code layout/design choices made,
• discuss how you handled the OOV, BOS, and EOS symbols (including any hyperparameters that could affect their use and the proper inclusion or exclusion from computing the probability distributions),
• give a detailed discussion in your writeup of how you verified your implementations are correct (this can be concise),
• show that you can train a model and compute dev perplexity results by completing the following table (see Perplexity). Your perplexity results do not have to be optimized.

<table>
<thead>
<tr>
<th>Basic Model Type</th>
<th>N-gram</th>
<th>Hyperparameters</th>
<th>Dev ppl.</th>
<th>Dev ppl. (ngram_lm_eval)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mle</td>
<td>1</td>
<td>???</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>???</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td>laplace</td>
<td>1</td>
<td>$\gamma = ???, ???$</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$\gamma = ???, ???$</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td>backoff</td>
<td>1</td>
<td>$\delta_n = ???, ???$</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$\delta_n = ???, ???$</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

6. (20 points) Use the code and scripts you’ve developed over Q3–5 to perform well-controlled experimentations of these models. Train and evaluate each of these eight models. The MLE models you can do just once, but for the others experiment with at least three different hyperparameter configurations. This will give you (at least) 14 development results (2 MLE models + 4 non-MLE models with 3 configurations each).

From this experimentation, create and turn in a report where you

• Document how you used the data splits properly to do your experimentation.
• Identify the best configuration, including value of $N$ and parameter values, for each MLE, laplace, and backoff.
• Evaluate these three best-in-class models (the best MLE, the best Laplace, the best backoff) on the testing set. This gives you three different test set results.
• Present the results, both internal dev and final testing, in readable formats, e.g., in tables or plots.
• 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.

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 two new datasets. One of the test sets will be drawn from the Universal Dependencies dataset; 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 grade.

You’ll need to tune your hyperparameters. For this assignment, “tuning” means trying different configurations of hyperparameters. For example, for each value $N$, for laplace try different values of $\gamma$.
and for backoff try different values of $\delta_n$ (or $\epsilon_n$). What’s listed here is not the only tuning you can do; it’s just an example. See Appendix B for additional ideas or examples.

7. (30 points) In this final question, you will implement a language identification system: given a sentence, identify (from a small [2-3 size] set of languages) the likely language that sentence is written in. You will develop and evaluate your system on languages from the Universal Dependencies dataset.

- **Your Task** Your system will be based upon the noisy channel model of classification: given a sentence $s$ tokenized into $S$ words $w_1, \ldots, w_S$, you will predict which language $l$ that $s$ is most likely to be in according to the language with the highest posterior probability:

\[
\hat{l} = \arg \max_l p(l \mid w_1, \ldots, w_S) \tag{9}
\]

\[
= \arg \max_l p^{(n)}(w_1, \ldots, w_S \mid l) p(l) \tag{10}
\]

\[
= \arg \max_l \log p^{(n)}(w_1, \ldots, w_S \mid l) + \log p(l). \tag{11}
\]

Here, $p^{(n)}(w_1, \ldots, w_S \mid l)$ is a language-conditioned $n$-gram probabilistic language model, and will be discussed more below.

Create and turn in a report in which you:

- Document internal progress and development by training on the training sets and evaluating on the development sets for each language you use. As in the previous question, this internal development must include some amount of experimentation.

- Discuss why the value $l$ that optimizes (11) is equivalent to the one that optimizes (9).

- Evaluate the best configuration on the testing set, and compare against a baseline language ID model.

- Present the results, both internal and final, in readable formats, e.g., in tables or plots, and clearly indicate in the writeup what model and model configurations resulted in the best test set performance.

- Summarize and analyze your results: what did you learn, and what do your experiments and results teach us about language ID?

- 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.

Turn in all code, including what is necessary for producing the plots. Also turn in the serialized version of the best performing system.

- **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.

- **System Specifics** The following elaborates on specifics for the system you are to build.

**Language Models** The likelihood of a sentence being generated by any language $l$ is given by a language-conditioned $n$-gram probabilistic language model: $p^{(n)}(w_1, \ldots, w_S \mid l)$. You may use any of the language models from the previous question, as long as they are non-trivial.
Prior Probabilities

The prior probability $p(l)$ represents the probability of a language occurring at all, without regard to any particular input sentence. Often, $p(l)$ is just the overall frequency of language $l$ observed in training: e.g., if 100 sentences were in language $A$ and 50 were in language $B$, then $p(l = A) = \frac{100}{100+50} = \frac{2}{3}$. Of course, the prior probability can itself be a smoothed estimate.

Languages to Experiment on

Overall, you must experiment with at least three different languages: you can do this either with two different pairs of languages (e.g., first pair of $A$ and $B$, and a second pair of $C$ and $D$); two different pairs of languages, with one language shared (first pair of $A$ and $B$, and a second pair of $A$ and $C$); or one set of three (or more) languages (e.g., three-way classification of $A$, $B$, and $C$). While you may experiment on fully distinct languages, e.g., English vs. Korean, you may also experiment on different subsets of a particular language. For example, how easy is it to distinguish UD_English-EWT from UD_English-GUM?

Proper Baseline Comparison

Include a proper baseline comparison. For this, you may implement the “most common” baseline. You can easily implement this by approximating the posterior via the prior probability:

$$
\hat{l} = \arg \max_l p(l \mid w_1, \ldots, w_S) \approx \arg \max_l p(l).
$$

Evaluation

Evaluate your predictions using precision, recall, and F1 score. While you may implement these yourself, you may also use 3rd party libraries. For instance, the Python library `sklearn` has functions `sklearn.metrics.precision_score` and `sklearn.metrics.recall_score`. The first argument to each function is a vector of ground truth labels (the actual labels); the second argument is a vector of predicted labels. By setting the parameter `average = 'micro'` or `average = 'macro'`, you can compute micro vs. macro scores. For example, the following snippet first computes micro recall, then macro precision:

```python
>>> gold = ['lang1', 'lang2', 'lang1', 'lang3']
>>> pred = ['lang3', 'lang2', 'lang1', 'lang3']
>>> sklearn.metrics.recall_score(gold, pred, average = "micro")
0.75

>>> sklearn.metrics.precision_score(gold, pred, average = "macro")
0.8333333333333337
```

Notice that the labels can be strings.

---

1 You of course can use other models. One we covered in class was **interpolated** models: $N$-gram models built from 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. Such an interpolation model will need mixing weights $\lambda$; see Appendix for details if interested.
Appendix A  Interpolation Model

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

\[
p^{(n)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) = \sum_{m=0}^{n} \lambda_{n,n-m} f^{(n-m)}(x_i|x_{i-n+1}, \ldots, x_{i-1})
\]

(13)

\[
= \lambda_{n,n} f^{(n)}(x_i|x_{i-n+1}, \ldots, x_{i-1}) + \lambda_{n,n-1} f^{(n-1)}(x_i|x_{i-n+2}, \ldots, x_{i-1}) + \lambda_{n,n-2} f^{(n-2)}(x_i|x_{i-n+3}, \ldots, x_{i-1}) + \ldots + \lambda_{n,0} \frac{1}{V}
\]

(14)

where \( f^{(0)} = \frac{1}{V} \) is the uniform distribution. So the bigram model interpolates 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

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

(15)

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

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

Appendix B  Tuning Language Models

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 \( \theta_t \) is through a grid search. Grid search selects a value for each \( \theta_t \), updates any necessary model parameters, and evaluates the model (on the development set), reporting both the end evaluation score and the values of \( \theta_t \). The final values for \( \theta_t \) are those that resulted in the best evaluation performance.

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). See Appendix C for more information about gradient-based tuning. As a reminder, you do not have to implement gradient-based tuning for this assignment. Grid search is perfectly acceptable for this assignment.
Appendix C  Tuning Parameters via Gradient Descent

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 $\Theta(s)$:

$$\Theta^{(s+1)} = \Theta^{(s)} - \rho_s \nabla_{\Theta} J(\Theta)|_{\Theta^{(s)}}. \quad (17)$$

The scaling factor is $\rho_s$: setting it is an entire area of active research. An effective learning rate is

$$\rho_s = \rho_0 \frac{1}{1 + \rho_0 \kappa s}, \quad (18)$$

where $\rho_0$ is an initial learning rate, and both $\rho_0$ and $\kappa$ 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 $x^*$ 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:

$$J(\Theta) = -\sum_{x,y} c(x, y) \log p_{\Theta}(y|x). \quad (19)$$

This is the negative log-likelihood.

Now, what can you tune? Integer-valued parameters, like $\epsilon_n$ 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 $\lambda_n$ 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 (17). However, you then define a projection map $\Pi$ 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 $\gamma_n$. While this can be effective, you must apply the chain rule in order to properly compute the gradient, $\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}$. A function like the sigmoid function, $\sigma(x) = \frac{1}{1+\exp(-x)}$, allows a reparametrization for the interpolation weights.

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.

---

2 See Leon Bottou’s “Stochastic Gradient Descent Tricks” for more.