# Assignment 2

CMSC 473/673 — Introduction to Natural Language Processing

Due Saturday October 21st, 2017, 11:59 PM

| **Item** | **Summary** |
|---|---|
| Assigned | Tuesday September 26th, 2017 |
| Due | Saturday October 21st |
| Topic | Maximum Entropy Modeling, Classification, and Distributed Representations |
| Points | 170 |

In this assignment you will understand and gain experience in both implementing and using maximum entropy (log-linear) models.

As with Assignment 1, 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 |
|---|---|---|---|---|---|
| **Points** | 40 | 10 | 30 | 30 | 60 |

As before, the first page provides 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 `a2`.

## Task List

1. Work through the online tutorial presented in class (`https://goo.gl/B23Rxo`). Answer the eight questions (a) through (h) presented in this assignment handout. You should think about, but do not have to answer, the questions posed directly in the tutorial (unless part of the eight questions mentioned above).

2. For a unigram maxent model with lexical features,

   (a) what are, *up to a constant*, the optimal weights $\theta_k^\star$ when optimizing the log-likelihood without regularization?

   (b) Is the optimal $\theta^\star$ guaranteed to be finite? If not, when will it not be finite?

3. Read and write a half page summary and review of Rosenfeld (1994). 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?

4. Using the file

   `/afs/umbc.edu/users/f/e/ferraro/pub/473-f17/a2/data/train.5k.processed.txt.gz`:

   (a) Prepare an 8 column table for a character *unigram* maxent model with lexical features, computing, for each feature, the observed feature count, and under three different parameter settings, the expected feature count and a finite difference approximation to the gradient.

   (b) Prepare an 8 column table for a character *bigram* maxent model with lexical bigram and unigram backoff features, computing, for each feature, the observed feature count, and under three different parameter settings, the expected feature count and a finite difference approximation to the gradient.

   Print all floating point numbers with at least 4 decimal places of precision.

   (This question leads into, and is meant to ease debugging, the next.)

5. Perform and write up well-controlled examinations of three classes of posterior classification 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 or model configurations (including what features are defined, any regularization, and how the prior distribution, if any, is estimated). When internal development is done, you are to pick the best configuration 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.

   Turn in all code, including what is necessary for producing the plots.

## Full Questions

1. (**40 points**) Work through the online tutorial at `https://goo.gl/B23Rxo`.[1] **It will take several hours to work through the lessons.** In the instructions box for each lesson, there are questions posed; you should think about them as you work through the tutorial, but you do **not** have to turn in your responses to them. Answer, and turn in your responses for, the following eight questions.

   (a) Answer the four questions of lesson 2.

   (b) Describe how the "log-likelihood" and "matching" games (defined in lessons 1 and 3) are related. When you make a good move in the log-likelihood game, does it seem to help or hurt the matching game? How about vice-versa?

   (c) Why is the added feature in lesson 4 important? Was it the only additional feature that could have been added? If so, explain; if not, what was another feature?

   (d) Compare and contrast the intended effects of $\ell_2$ vs. $\ell_1$ vs. no regularization in lesson 8.

   (e) Summarize, in your own words, the tradeoff(s) between the number of "tokens" seen and $C$, the regularization coefficient.

   (f) Discuss how, and explain why, lesson 9 and 10 differ.

   (g) Describe the end effect(s) of adding features for contexts, as shown in lesson 14. Compare the setup and end results, including optimal log-likelihood and the overall sensitivity of the objective to individual feature weights, in 14 with that of 15. Discuss the (real-world) implications of this. When using log-linear models, what should you do (or not do)?

   (h) In the bigram modeling (16) or classification (17) lessons, how well can your model fit the observed data? What are some of the highest and lowest weighted features? How does your answer change as you change the amount of observed data, the regularization, or both?

   You should play with both lessons, but you only need to answer this question (part 1h) for either lesson 16 *or* lesson 17. Please indicate which lesson you're referring to. (Of course, you *are* welcome to answer for both lessons.)

   You may work through and discuss the lessons with others. However, you must answer and write up responses to the above on your own.

2. (**10 points**) Consider a unigram maxent model $p(z) \propto \exp(\theta^\intercal f(z))$, defined over $V$ types. Define $V$ binary features $f_k$ as

$$f_k(z) = \begin{cases} 1 & \text{if } z = k \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

   Therefore, there are $V$ different features; correspondingly, there are $V$ different weights. Features like these are called lexical features. Notice that we're indexing both the feature vector and feature weights by the vocabulary items.[2]

   (a) If you trained this model's log-likelihood without any regularization, what are, *up to a constant*, the optimal weights $\theta_k^\star$?

   (Hint 1: think about other unigram models we've covered.)

[1] This is a shortened URL for `https://www.csee.umbc.edu/courses/undergraduate/473/f17/loglin-tutorial/`.

[2] For example, our vocabulary is over the words "foo" and "bar." There are two features, $f_{\text{foo}}(z)$ and $f_{\text{bar}}(z)$, that act accordingly: $f_{\text{foo}}(\text{foo}) = 1$, $f_{\text{foo}}(\text{bar}) = 0$, $f_{\text{bar}}(\text{bar}) = 1$, and $f_{\text{bar}}(\text{foo}) = 0$.

(Hint 2: the constant in "up to a constant" may be defined with respect to some fixed training data.

(b) Is the optimal $\theta^\star$ guaranteed to be finite? If not, when will it not be finite?

3. (**30 points**) Read and write a half page summary[3] and review of Rosenfeld (1994). It is available on GL at

   `/afs/umbc.edu/users/f/e/ferraro/pub/473-f17/a2/rosenfeld-1994.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{rosenfeld1994hybrid,
  title={A Hybrid Approach to Adaptive Statistical Language Modeling},
  author={Rosenfeld, Ronald},
  booktitle={Proceedings of the Workshop on Human Language Technology},
  pages={76--81},
  year={1994},
  organization={Association for Computational Linguistics}
}
```

   (While many of the acronyms are defined, a couple aren't. CSR, mentioned at the beginning of Section 5, stands for "continuous speech recognition." ARPA—the Advanced Research Projects Agency—was a U.S. government agency that provided some research funding; it was the predecessor to the current DARPA—the Defense Advanced Research Projects Agency.)

4. (**30 points**) This question sets you up for question 5, where you will implement and apply maxent models for classification. Some people may find it easier to answer this question by first implementing a general conditional maxent model that can be used in this question and in question 5; other people may find it easier to first work through this question without any of the overhead of 5. It is your choice how to approach this question and 5. You *may* use code written for this question in 5, and vice versa.

   In order to train (optimize log-likelihood) and use a conditional maxent model $p(y|x) \propto \exp(\theta^\mathsf{T} f(x, y))$, you need to be able to: (i) evaluate $p(y|x)$ (or an equivalent form such as $\log p(y|x)$), (ii) evaluate the log-likelihood $J$, (iii) compute the observed feature counts, and (iv) compute the expected feature counts. Note that those last two are needed to compute the gradient of log-likelihood.

   Properly computing the gradient is crucial, but it can be tricky to get right. One method for debugging gradients is through *finite differences*. If you are familiar with finite difference checks, you may skip the following explanatory block.

   > **Tutorial: Finite Difference Approximation**
   >
   > To start, let's first look at derivatives. A derivative $\frac{df(x)}{dx}$ gives the *instantaneous* rate of change of the function $f$ at each point. Mathematically, you may have seen derivatives as
   >
   > $$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$
   >
   > If you have implemented the derivative in a function $g(x)$, such that $g(x)$ is **intended** to compute $\frac{df(x)}{dx}$, a finite difference approximation simply approximates $g(x)$ by setting $h$ to

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

some sufficiently small value (such as 0.000001). That is, it checks (for a variety of values $x_0$) to make sure that, for a sufficiently small value $h$,

$$g(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

To extend the finite difference check to multivariable functions $F(\theta)$ (i.e., gradients), remember that the gradient is just a vector of partial derivatives, where a partial derivative $\frac{\partial F}{\partial \theta_i}$ is just the derivative of $F$ with respect to the component $\theta_i$ where you treat all other components of $\theta$ ($\theta_{j \neq i}$) as constant. Mathematically, we can define partial derivatives as

$$\begin{aligned}\frac{\partial F(\theta)}{\partial \theta_i} &= \lim_{h \to 0} \frac{F(\overbrace{\theta_1, \ldots, \theta_{i-1}}^{\text{do not adjust any other components}}, \overbrace{\theta_i + h}^{\text{only add } h \text{ to the component } \theta_i}, \overbrace{\theta_{i+1}, \ldots, \theta_K}^{\text{do not adjust any other components}}) - F(\theta)}{h} \\ &= \lim_{h \to 0} \frac{F(\theta + (0, 0, \ldots, h, \ldots, 0)) - F(\theta)}{h}.\end{aligned}$$

Then, if we have a function $G(\theta)$ that computes $\nabla_\theta F$, i.e., the gradient of $F$, we can apply our finite difference check to each component,

$$\begin{aligned}G(\theta)_1 &= \frac{\partial F}{\partial \theta_1} \approx \frac{F(\theta + (h, 0, \ldots, 0)) - F(\theta)}{h} \\ G(\theta)_2 &= \frac{\partial F}{\partial \theta_2} \approx \frac{F(\theta + (0, h, \ldots, 0)) - F(\theta)}{h} \\ &\ldots \\ G(\theta)_K &= \frac{\partial F}{\partial \theta_K} \approx \frac{F(\theta + (0, 0, \ldots, h)) - F(\theta)}{h}.\end{aligned}$$

If you're wondering if we could get around the hassle of computing the gradient when we could approximate it with finite differences: while finite differences serve as a useful debugging method, in practice (in an iterative optimization routine) they can be very noisy and adversely affect the convergence of the optimization itself.

For this question, use the compressed text file at

`/afs/umbc.edu/users/f/e/ferraro/pub/473-f17/a2/data/train.5k.processed.txt.gz`.

This is a postprocessed version of the training data from assignment 1: (i) all characters have been lowercased; (ii) only alphanumeric (a-z, 0-9), dashes (-), spaces (ASCII decimal code 32 decimal) have been kept; and (iii) all whitespace, including newlines, has been collapsed to a single space (ASCII decimal 32). Note that this file has only 1 line in it.

Define your outcome vocabulary $V$ over each of these *characters*, including space; be sure to include an end-of-sentence and out-of-vocabulary symbols. Define your context vocabulary $C$ over these characters (including space), a beginning-of-sentence symbol and the out-of-vocabulary symbol. You may want to let the beginning-of-sentence symbol be "B," the end-of-sentence symbol be "E," and the out-of-vocabulary symbol be "U" (for unknown). With this definition, there should be 40 (outcome) vocabulary items and 40 (context) vocabulary items; you should verify this on your own, but you do not have to prove it in any way.

**Hint**: if you get stuck, you can debug against the online tutorial. For question 4a you can look at lesson 1, and for question 4b you can look at lesson 12.

(a) To start, let's just look at a unigram character maxent model $p(y) \propto \exp(\theta^\intercal f(y))$ with lexical features defined over the outcome vocabulary $V$, as in question 2. That is, given the above file (sequence of characters $y_1, y_2, \ldots, y_S$), the log-likelihood is computed as

$$J(\theta) = \frac{1}{S} \sum_i \log p(y_i).$$

Produce a *comma* separated file with eight columns; turn in the file and indicate its name in your writeup.

Let $J$ be the log-likelihood. The eight columns are:

(i) the feature name $\beta$ (e.g., how it's indexed, or the $k$ in equation 1);

(ii) the observed feature count for $\beta$;

(iii) the expected feature count for $\beta$, when $\theta_\beta = 0.0$,

(iv) a finite difference approximation of $\frac{\partial J}{\partial \theta_\beta}$ when $\theta_\beta = 0.0$,

(v) the expected feature count for $\beta$, when $\theta_\beta = 1.0$,

(vi) a finite difference approximation of $\frac{\partial J}{\partial \theta_\beta}$ when $\theta_\beta = 1.0$,

(vii) the expected feature count for $\beta$, when

$$\theta_\beta = \tanh\left(\frac{\text{charcode}(\beta) - 74}{128}\right),$$

(viii) a finite difference approximation of $\frac{\partial J}{\partial \theta_\beta}$ when $\theta_\beta$ is computed as in the previous column.

This table is trying to help you debug the gradient computation. So what are you looking for here? You want the difference between the observed feature count and the expected feature count to be approximately equal to the corresponding finite difference approximation.

**For grading**, we will be looking at the observed and expected feature counts. The finite difference approximation is for your benefit; you may decide what to set $h$ as (indicate the value you chose in your write-up).

To clarify the last setting of $\theta_\beta$, the $\text{charcode}(\beta)$ function returns the *decimal* ASCII value of letter $\beta$. For example, the weight for feature "a" is $\theta_{\texttt{a}} = \tanh(\frac{97-74}{128}) \approx 0.1778$, while the weight for a space character is $\theta_{\texttt{SPACE}} = \tanh(\frac{32-74}{128}) \approx -0.3168$. In Python, you can use the `ord` function to get the character code and `numpy` to access tanh:

```
$ python
Python 2.7.13 |Anaconda 4.4.0 (64-bit)| ...
>>> ord(' ')
32
>>> ord('a')
97
>>> import numpy
>>> numpy.tanh((97.0 - 74)/128)
```

```
0.17777826153566401
>>> def get_theta(letter):
...     return numpy.tanh((float(ord(letter)) - 74.0)/128.0)
...
>>> get_theta('a')
0.17777826153566401
>>> get_theta(' ')
-0.31683500112336604
>>>
```

If you represent beginning-of-sentence, end-of-sentence, and the unknown symbols with B, E, and U, respectively, the `get_theta` as defined will be sufficient. Otherwise, you'll need to slightly adjust the definition according to your representation.

When printing, please ensure **all floats have at least four decimal points of precision**. The order of features does not matter. You may indicate the "space" name as something like `SPACE`, as shown above.

(b) Now let's move on to a conditional maxent model $p(y|x)$; this models character bigrams. The features will still be lexical, but at both the unigram and bigram level: define features

$$f_{i,j}(x,y) = \begin{cases} 1 & \text{if } i = x \text{ and } j = y \\ 0 & \text{otherwise,} \end{cases} \qquad f_k(x,y) = \begin{cases} 1 & \text{if } k = y \\ 0 & \text{otherwise.} \end{cases}$$

Note that $j$ and $k$ should be defined over the outcome vocabulary, while $i$ should be defined over the context vocabulary.

Repeat what you did above in this new model. This time, the log-likelihood is computed as

$$J(\theta) = \frac{1}{S} \sum_i \log p(y_i|y_{i-1}).$$

Produce a *comma* separated file with eight columns; turn in the file and indicate its name in your writeup. The eight columns are:

(i) the feature name $\beta$ (e.g., how it's indexed, either the $k$ or the $i, j$ pair);

(ii) the observed feature count $\beta$;

(iii) the expected feature count for feature $\beta$, when $\theta_\beta = 0.0$,

(iv) a finite difference approximation of $\frac{\partial J}{\partial \theta_\beta}$ when $\theta_\beta = 0.0$,

(v) the expected feature count for feature $\beta$, when $\theta_\beta = 1.0$,

(vi) a finite difference approximation of $\frac{\partial J}{\partial \theta_\beta}$ when $\theta_\beta = 1.0$,

(vii) the expected feature count for feature $\beta$, when

$$\theta_k = \tanh\left(\frac{\text{charcode}(k) - 74}{128}\right),$$

and

$$\theta_{i,j} = \tanh\left(\frac{\text{charcode}(j) - \text{charcode}(i)}{128}\right),$$

(viii) a finite difference approximation of $\frac{\partial J}{\partial \theta_\beta}$ when $\theta_\beta$ is set in the previous column.

In Python, you can use the following definition:

```
>>> def get_theta2(i, j):
...     return numpy.tanh((float(ord(j)) - float(ord(i)))/128.0)
...
>>>
```

As before, please ensure **all floats have at least four decimal points of precision**, the order of features does not matter, and you may indicate the "space" portion of any name involving the space character as something like `SPACE`.

5. (**60 points**) In this question, you will use maximum entropy models in order to perform language ID.

*Code-switching* occurs when a person switches among multiple languages in the course of a single conversation. For instance, in the following tweet, a user switches between English and (Latinized) Hindi:

| @timesofindia | @ArvindKejriwal | what | have | u | been | doing | snooping | on |
|---|---|---|---|---|---|---|---|---|
| univ | univ | en | en | hi | en | en | en | hi |
| yadav | and | prashant | nasihat | bahut | dete | ho | aina | dekho |
| hi | en | hi | hi | hi | hi | hi | hi | hi |

The tweet is split over two lines: in the normal font is the tokenized text, and in the `constant width` font are the per-word language labels (below the corresponding token). Words tagged with `en` are marked as English, those tagged with `hi` are marked as Hindi, and language-independent tokens, like usernames, are tagged with `univ`.

**Your Task** Your task is to perform well-controlled examinations of the following models (described below) on per-word language ID in code-switched social media posts. Specifically,

- Document internal progress and development by training on the training set (optimizing the objective via gradient descent/ascent) and evaluating on the development set (performing MAP classification).
- This internal development **must** include some amount of tuning of hyperparameters or model configurations (including what features are defined, any regularization, and how the prior distribution, if any, is estimated).
- When internal development is done, you are to pick the best configuration 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, and that your gradient computations are correct.

Turn in all code, including what is necessary for producing the plots. 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.

**Models** Implement the following per-word classifiers, which identify the language of individual words in text (social media posts):

(i) A maximum a posteriori baseline classifier of your choice. (As an example, a classifier that used a Laplace smoothed bigram likelihood and uniform prior would be sufficient. You are welcome to use any of the models from assignment 1.)

(ii) A maximum a posteriori classifier $p(l|y) \propto p(y|l)p(l)$ , where the likelihood $p(y|l)$ is a maximum entropy character $n$-gram model, and $p(l)$ is a prior estimate of the probability of a label $l$. The value $n$ you use is up to you, but you should use at least bigrams (i.e., $n \geq 2$). You should train the character $n$-gram model to optimize (regularized) log-likelihood of each word (represented as a sequence of $n$-grams), given the word's language label $l$.

(iii) A maximum a posteriori classifier $p(l|z) \propto \exp(\phi^\intercal g(l, z))$, directly modeled with a conditional log-linear distribution. You should train the model to optimize (regularized) log-likelihood of a word's label $l$, given that word and any extra word or subword information $z$. Treating each word of every post independently from one another, the log-likelihood of $M$ label/word pairs is

$$J(\phi) = \sum_{m=1}^{M} \log p(l_m|z_m) = \left(\sum_{m=1}^{M} \phi^\intercal g(z_m, l_m)\right) - \left(\sum_{m=1}^{M} \log\left(\sum_{l'} \exp(\phi^\intercal g(z_m, l'))\right)\right).$$

(For instance, $z$ could be both the word and its part-of-speech.)

For both maxent model types, you should **experiment**, at least internally on dev data, with features that are beyond lexical. See the "Features and Auxiliary Data" portion below.

**Data** The training, development, and testing data are available as gzipped JSON files in the directory

`/afs/umbc.edu/users/f/e/ferraro/pub/473-f17/a2/data/cs/`.

Each file is a dictionary; the keys are instance IDs,[4] and the values are the actual tagged posts.[5] Each post is represented as a list of tagged tokens, where each tagged token is a dictionary. Within each tagged token dictionary, there are three keys: "text" provides the word itself, "lang" indicates the language label of the word, and "pos" is a part-of-speech tag of the word. For example, the first five words of the above tweet, which has ID 238, are represented as

```
{
  ...
  "238": [
    {
      "lang": "univ",
      "text": "@timesofindia",
      "pos": "@"
    },
```

[4] The instance ID isn't actually needed for the task; it's provided simply to give you an easy way to refer to posts *if* you need to.

[5] The data are from `http://amitavadas.com/Code-Mixing.html`.

```
    {
      "lang": "univ",
      "text": "@ArvindKejriwal",
      "pos": "@"
    },
    {
      "lang": "en",
      "text": "what",
      "pos": "PR_PRP"
    },
    {
      "lang": "en",
      "text": "have",
      "pos": "V_VM"
    },
    {
      "lang": "hi",
      "text": "u",
      "pos": "PR_PRP"
    },
    ...
  ],
  ...
}
```


Notice this is a multi-class classification problem. Further, the data contains code-switched English-Hindi, English-Bengali, and English-Telugu. (There are some "universal" language labels as well.)

**Features and Auxiliary Data** *The features you use in either maxent models are up to you.* Often, you can do moderately well with lexical features. However, your features do not have to be only lexical. They can also capture more general information about the items being featurized. For instance:

- You could also use a model from the first assignment to define a feature. The most straightforward is to have a feature return $\log p(\beta|\alpha)$, where the probability model $p$ is one of the smoothed language model (but training on characters!).
- Just as there are word embeddings, there can also be character embeddings: there are some available for you to use in

  `/afs/umbc.edu/users/f/e/ferraro/pub/473-f17/a2/data/embeddings/`.

  (Note: there may not be an embedding for every possible character you see.)

  In a character bigram model $p(\beta|\alpha)$, one feature could be the cosine similarity between embeddings for characters $\alpha$ and $\beta$:

$$\text{cosine}(\alpha, \beta) = \frac{\sum_i \omega_{\alpha,i}\omega_{\beta,i}}{\sqrt{\sum_i \omega_{\alpha,i}^2}\sqrt{\sum_i \omega_{\beta,i}^2}},$$

  where $\omega_\alpha$ and $\omega_\beta$ are embedding vectors of the characters $\alpha$ and $\beta$. If there are $C$ context items and $V$ outcome items, this adds $CV$ features.

- Alternatively, you could define a different feature for each component $\omega_{\beta,i}$: if the outcome vocabulary has $V$ items and each embedding is 200 dimensions in size, then this adds $200V$ features.

Remember though that the $n$-gram maxent model is *generative*: it must model *just* a sequence of characters (given a label), and the way that it models each character sequence is via character $n$-grams. Therefore, unless you also extend the $n$-gram model to generate part-of-speech tags, your maxent $n$-gram model should not use the part-of-speech tags as any part of the features. However, your conditional maxent classifier (the third model type) *can* use part-of-speech tags. It may also use features based on the entire *word*, such as word embeddings, in addition to looking at internal $n$-grams. There are some word embeddings in the above folder you can use.

You may use other embeddings as you see fit. However, these embedding files tend to be very large, even when compressed. If you do use other embedding files, please email the instructors so we can coordinate getting access to them easily.

**Gradient Ascent/Descent** *It is up to you if you write your own optimizer or use an existing one.* You will need to optimize the parameters ($\theta$ or $\phi$) with respect to their respective objectives. Recall that gradient descent (ascent) proceeds iteratively. In gradient *descent*, the aim is to minimize an objective. 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 $\Theta^{(s)}$:

$$\Theta^{(s+1)} = \Theta^{(s)} - \rho_s \nabla_\Theta J(\Theta)|_{\Theta^{(s)}}. \tag{2}$$

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

The above formulation considered an objective to minimize. However, the log-likelihood objectives should be maximized. To reconcile this, simply minimize the negative log-likelihood. If you do minimize the negative log-likelihood, don't forget to adjust the gradient ($\nabla - F = -\nabla F$).[6]

**Batch Training vs. Stochastic (Online) Training** Setting the scaling factor $\rho$ can be tricky. One of the things that makes it tricky is how you "batch" your data during training. *It is up to you how to batch your data.*

In full batch training, you compute your objective and gradient over the entire dataset. This can give robust statistics and updates. However, because you have to iterate and perform computations over the entire dataset just to do a single update (i.e., to apply equation 2 once), full batch training can be computationally expensive. **For full batch training, I highly recommend using a 3rd party optimization routine.** These 3rd party modules will take care of properly setting the scaling factor and determining convergence for you.

In mini-batch training, you compute your objective and gradient over a portion of the dataset. First, make sure that data is in sufficiently random order. Then, for a batch of size $B$, you iteratively take $B$ training instances, compute your objective and gradient over those instances, and apply the update in 2. Because each time you're updating you're doing so based on fewer examples, mini-batch training can be a bit noisy; it can also be sensitive to the order in which it sees certain examples.[7] **For mini-batch training, it will be easiest to write your own gradient descent/ascent.** However, in mini-batch training you will need to rescale the regularizer in the objective: if there are $M$ instances total, and your batch is of size $B$, you must multiply the regularizer (and its gradient) by $\frac{B}{M}$.

[6] If you're writing your own gradient optimization routine, then you are free to maximize or minimize. On the other hand, most 3rd party libraries *minimize* functions.

[7] If in a character bigram model, it sees all sequences starting with an "a" (e.g., aa, ab, ac) before any other sequence (e.g., ba, cd, fe), then the update could be heavily biased to prefer instances starting with "a."

If you use an external library to optimize, that's fine: while it can be straightforward to write your own gradient descent optimizer, they can get complicated quickly. You are allowed to use an external optimization library to perform gradient-based tuning. In Python, you can use the `scipy.optimize` module. Assume that you've implemented your objective and its gradient as `f` and `df`, which both take as their primary arguments `x`, a vector of size $K$. The objective returns a single real number, while the gradient returns a vector of size $K$. You can use the `scipy` submodule as:

```
optimize_result = scipy.optimize.minimize(
  f, ## the objective function
  numpy.zeros(K), ## the initial setting for the parameters to optimize
  method = 'L-BFGS-B', ## the precise optimization routine
  jac = df ## the gradient function
)
```

However, regardless of what (if any) library you use, you **must** derive and *implement* the gradient.

If you write your own optimizer, you will need to update your scaling factor. While this is an area of active research, an effective[8] learning rate is

$$\rho_s = \frac{\rho_0}{1 + \rho_0 \frac{\kappa}{M} s}, \tag{3}$$

where $\rho_0$ is an initial learning rate, and both $\rho_0$ and $\kappa$ are some constants you can set. Setting $\kappa = 1$ and $\rho_0$ to either $0.01$ or $0.1$ are generally considered reasonable, though.

**Predicting** Prediction should be done on a per-word basis. That is, each word's language should be predicted independently and separately from one another.

**Evaluation** Evaluate your predictions using macro-precision, macro-recall, micro-recall, and micro-precision. While you may implement these yourself, you may also use 3rd party libraries. For instance, the Python library `sklearn` (available through the Anaconda build) 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:

```
>>> gold = ['en', 'hi' ,'en' ,'univ']
>>> pred = ['univ','hi','en' ,'univ']
>>> sklearn.metrics.recall_score(gold, pred,
                                 average = "micro")
0.75

>>> sklearn.metrics.precision_score(gold, pred,
                                    average = "macro")
0.83333333333333337
```

Notice that the labels can be strings.

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