CS 475/675 Machine Learning: Homework 6
Graphical Models and Inference
Programming Assignment

Due: Monday, December 7, 2020, 11:59 pm

35 Points Total Version 1.2

Make sure to read from start to finish before beginning the assignment.

This assignment must be completed individually. No partners allowed.

1 Introduction

In this assignment you will be exploring a technique for natural language understanding known as Topic Modeling. You will be implementing a Variational Inference method that uses the Mean Field approximation for learning the parameters of the probabilistic LDA graphical model.

We have provided Python code to serve as a test-bed for your algorithms. We'll use a similar coding framework to those we've used in previous assignments. You will fill in the details. Search for comments that begin with `TODO`; these sections need to be written. Your code for this assignment will all be within the `models.py`; do not modify any of the other files.

2 Data

We will be using State of the Union (SotU) speeches given by US presidents[1]. SotU addresses have been delivered annually by the president since 1790. The purpose of these speeches is to summarize the affairs of the US federal government. Typically, the SotU addresses given by a newly inaugurated president contain a different tone. Your task in this assignment is to uncover topics important to US federal affairs over the years.

The corpus contains $D = 232$ documents, comprising years 1790-2020 (there were two SotU addresses in 1961 by Eisenhower and Kennedy). Each document d has a length N_d on the order of 10^3 words. The overall corpus vocabulary contains approximately 10^4 words. Like in homework 4, we have pre-processed the data for you using standard techniques from NLP, including tokenization, converting all words to lowercase, and removing stopwords[2], punctuation, and numbers.

[1] `https://en.m.wikisource.org/wiki/Portal:State_of_the_Union_Speeches_by_United_States_Presidents`

[2] `https://en.wikipedia.org/wiki/Stop_word`

Then the document-word matrix is constructed by computing the frequency of each word in each document. Finally, it is stored in a sparse `coo_matrix`[3] using a 32-bit integer format for space efficiency. See Section 4.1 for tips on working with `coo_matrix`.

3 Topic Modeling

Natural language is complex; modeling and extracting information from it requires breaking down language layer by layer. In linguistics, semantics is the study of meaning of words and sentences[4]. At the document level, one of the most useful techniques for understanding text is analyzing its topics. In this model, a topic is just a collection of words and an observed document is a mixture of topics. The key idea of the topic model is that the semantic meaning of a document is influenced by these latent topics.

3.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is a popular topic modeling technique for uncovering latent topics. LDA is a generative probabilistic model of a corpus (collection of text)[5][6]. In the LDA model, documents are represented as random mixtures over latent topics, whereas topics are characterized as distributions over words. Each topic defines a multinomial distribution over the vocabulary and is assumed to have been drawn from a Dirichlet $\phi_k \sim Dir(\beta)$. For each document d in the corpus, the generative story of LDA is as follows:

1. For each topic $k = 1...K$:
   (a) Choose $\phi_k \sim Dir(\beta)$. ϕ_k gives the probability that word w is in topic k.
2. For each document $d = 1...D$:
   (a) Choose $\theta_d \sim Dir(\alpha)$. θ_d gives the probability that document d has topic k.
   (b) For each word w_i in document d:
      i. Choose a topic $z_{di} \sim$ Multinomial(θ_d)
      ii. Choose a word $w_i \sim$ Multinomial$(\phi_{z_{di}})$

where α and β are hyperparameters of the two Dirichlet distributions. These are typically small values chosen to be < 1. For simplicity in this assignment, we will assume symmetric priors on both Dirichlet distributions, but this requirement can be relaxed. Given these two parameters, the joint distribution of the topic model is expressed:

$$P(w, z, \theta, \phi|\alpha, \beta) = \prod_{k=1}^{K} P(\phi_k|\beta) \prod_{d=1}^{D} P(\theta_d|\alpha) \prod_{i=1}^{N_d} P(z_{di}|\theta_d) P(W_i|\phi_{z_{di}}) \quad (1)$$

[3]Documentation: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html`

[4]If you're interested in learning more about this hierarchy of language structure, check out Professor Kevin Duh's lecture "Linguistics 101" from NLP Fall 2019.

[5]Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent Dirichlet Allocation." Journal of Machine Learning Research. 3, Jan (2003): 993-1022.

[6]Hoffman, M., Blei, D., Bach, F. Online learning for latent Dirichlet allocation. In Neural Information Processing Systems (2010).

The Dirichlet distribution[7] is a member of the exponential family of distributions, whose conjugate prior is the multinomial distribution. This means that when computing the MLE or MAP, we can simplify our expressions to the addition of sufficient statistics. Additionally, we have two sparsity requirements for the topic model: (1) document sparsity and (2) word sparsity. Each document d should contain only a small number of topics and each topic k should be determined by a small subset of words in the total vocabulary. Dirichlet distributions encode these sparsity requirements. The Dirichlet distribution is paramaterized by a vector of concentration parameters α. As $\alpha \to 0$, the multinomials drawn from the Dirichlet tend to be more sparse.[8]

This generative story is concisely represented using plate notation:

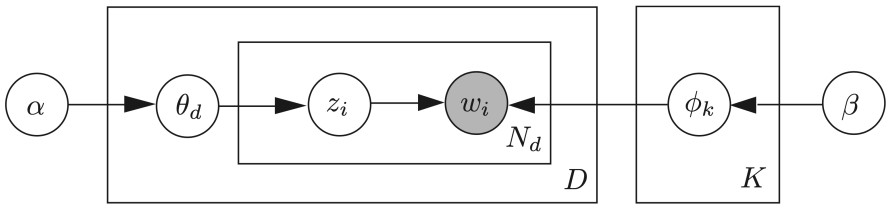

Figure 1: Three-layer hierarchal Bayesian LDA graphical model. Here, the use of plate notation represents replication. That is, document d is repeated D times in the corpus, the word tokens w_{di} are repeated N_d times in each document d, where N denotes the number of words in the document. There is a fixed vocabulary size of V and K topics. The document-specific topic proportion θ_{dk} generates a topic label $z_{dik} \in \{0, 1\}$ such that $\sum_{k=1}^{K} z_{dik} = 1$. These topic labels z_{di} then generate each observed word token w_i in the document d based on the topic-specific multinomial distribution ϕ_{kv} over all words in the vocabulary. The multinomial distributions θ_d and ϕ_k are generated by Dirichlet distributions with hyperparameters α and β, respectively.

LDA can learn which words in a text are associated with a specific topic k, expressed through their topic distribution ϕ_k. θ_d is a low-dimensional representation of document d in topic space, where z_{di} represents which topics generated a particular word w_i.

3.2 Inference

In order to use LDA, we must learn the latent parameters z, θ, and ϕ using inference. To do this, we must compute the posterior distribution of the latent variables Z, Θ, and Π given the the model parameters α and β, and observed data W:

$$P(Z, \Theta, \Phi | W, \alpha, \beta) = \frac{P(W, Z, \Theta, \Phi | \alpha, \beta)}{P(W | \alpha, \beta)} \tag{2}$$

Unfortunately, this posterior is intractable to compute in general. To normalize the distribution, we must marginalize over the latent variables. Specifically, examining the

[7] `https://en.wikipedia.org/wiki/Dirichlet_distribution`

[8] `http://www.cs.cmu.edu/~dbamman/notes/dirichletConcentration.pdf`

form of $P(W|\alpha, \beta)$ we have:

$$P(W|\alpha,\beta) = \int_{\Phi}\int_{\Theta}\sum_{Z} P(W, Z, \Theta, \Phi|\alpha, \beta)d\Theta d\Phi \tag{3}$$

$$= \int_{\Phi} P(\Phi|\beta)\int_{\Theta} p(\Theta|\alpha)\sum_{Z} P(Z|\Theta)P(W|Z, \Phi)d\Theta d\Phi \tag{4}$$

Computing the posterior exactly is intractable because of the coupling between Θ and Φ in the summation over latent topic assignments. In other words, we cannot make statements about maximizing a particular term since all the latent variables are coupled together.

Since we cannot compute this posterior exactly, we turn to approximate inference methods. Specifically, in this assignment you will be using variational inference to approximate the posterior distribution.

3.3 Mean Field Variational Inference

In Variational Inference, we design a family of tractable distributions $\mathcal{Q}$, which have parameters that can be tuned to approximate the true posterior. To obtain $\mathcal{Q}$, we will use the Mean Field approximation, which approximates the complex $p(Z, \Theta, \Phi|W, \alpha, \beta)$ with a fully factorized approximation. For LDA, this variational distribution is:

$$\mathcal{Q}(Z, \Theta, \Phi|\pi, \gamma, \lambda) = \prod_{k=1}^{K} Dir(\phi_i|\lambda_i)\prod_{d=1}^{D} q_d(\theta_d|\gamma_d)\prod_{i=1}^{N_d} q_d(z_{di}|\pi_{di}) \tag{5}$$

where λ_i, γ_d, and π_{di} are the free variational parameters for the variational distribution $q_d(\cdot)$ for document d.

The graphical model representation of this factorized variational distribution is shown in Figure 2. Notice that in this graphical mode, some of the nodes and edges were removed, which removes the problematic coupling that caused the intractability in the true posterior.

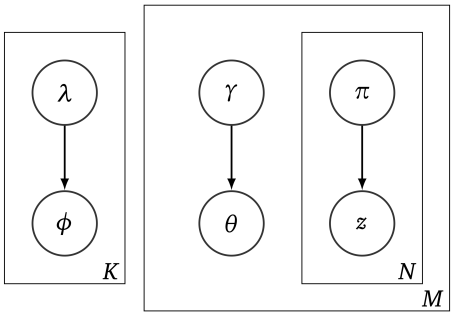

Figure 2: Graphical model for factorized variational distribution for smoothed LDA from Blei et al. Here, they use M to represent the number of documents and N the number of words in each document (where we use D and N_d before).

With a tractable family of probability parameters specified, we can optimize the free variational parameters so that the variational distribution best approximates the true

posterior. These parameters are chosen so that the KL-divergence between the variational distribution and true posterior is minimized and the ELBO (variational free energy) is maximized:

$$(\lambda^*, \gamma^*, \pi^*) = \arg\min_{\lambda,\gamma,\pi} KL(q(z, \theta, \phi|\lambda, \gamma, \pi)||p(z, \theta, \phi|w, \alpha, \beta)) \tag{6}$$

$$= \arg\max_{\lambda,\gamma,\pi} \mathbb{E}_q[\log p(w, z, \theta, \phi|\alpha, \beta] - \mathbb{E}_q[\log q(z, \theta, \phi] \tag{7}$$

Solving this optimization problem, we can derive the following variational parameter updates[9] that minimize the KL divergence from p to q:

$$\pi_{dik}^{(t+1)} \propto \exp\left(\mathbb{E}_q[\log \theta_{dk}] + \mathbb{E}_q[\log \phi_{kw}]\right) \tag{8}$$

$$\gamma_{dk}^{(t+1)} = \alpha + \sum_{i=1}^{N_d} \pi_{dik}^{(t)} \tag{9}$$

$$\lambda_{kv}^{(t+1)} = \beta + \sum_{d=1}^{D} \sum_{i=1}^{N_d} \mathbb{I}(x_{di} = v)\pi_{dik}^{(t)} \tag{10}$$

where

$$\mathbb{E}_q[\log \theta_{dk}] = \Psi(\gamma_{dk}^{(t+1)}) - \Psi\left(\sum_{i=1}^{K} \gamma_{di}^{(t+1)}\right) \tag{11}$$

$$\mathbb{E}_q[\log \phi_{kw}] = \Psi(\lambda_{kw}^{(t+1)}) - \Psi\left(\sum_{v=1}^{V} \lambda_{kv}^{(t+1)}\right) \tag{12}$$

where Ψ is the digamma function.[10]

Typically, you would use the EM algorithm to also learn the parameters α and β specific to each document and topic, respectively. For simplicity in this assignment, you will treat α and β as hyperparameters fixed at the start of inference.

3.4 Sufficient Statistics

Updating and storing all values in π_{dik} is computationally and space inefficient. Fortunately, since the Dirichlet distribution is a member of the exponential family, we can utilize sufficient statistics for updating λ[11]. Then, for updating γ, we can calculate the portion of π needed for that specific document d. **You should not explicitly calculate the entire π matrix**.

You will update λ based on the expected sufficient statistics

$$S(\lambda) \leftarrow \sum_{d=1}^{D} \sum_{i=1}^{N_d} \tilde{\pi}_{dik} \cdot x_{di}^T \tag{13}$$

where $\tilde{\pi}_d$ is the implicit computation of the variational parameter π. **This is not a normalized quantity. You will need to normalize by $\sum_{k=1}^{K} \tilde{\pi}_{dik}$.**

[9] For the complete derivation of these updates see `http://times.cs.uiuc.edu/course/598f16/notes/lda-survey.pdf`

[10] This is the first derivative of the $\log \Gamma$ function. For more details see: `https://en.wikipedia.org/wiki/Digamma_function`

[11] Mandt, Stephan, and David Blei. "Smoothed Gradients for Stochastic Variational Inference." ArXiv:1406.3650 [Cs, Stat], Nov. 2014.

3.5 Implementation Details

Optimization in the LDA model corresponds to:

1. Updating γ and π with λ fixed
2. Updating λ with π fixed

Both tasks are dependent on one another, so you will use Expectation-Maximization to iteratively optimize both parts of this optimization problem. Again, you won't be explicitly computing π in the E-step, and instead will compute sufficient statistics to speed up your training.

We provide the following pseudocode for one iteration to optimize the Variational parameters γ and λ (and implicitly π, by use of sufficient statistics) to make the Variational distributions as similar as possible (according to to the KL-Divergence, which is a metric to measure the difference between two probability distributions) to the true posterior $P(Z, \Theta, \Phi | W, \alpha, \beta)$.

You will first initialize the variational parameter λ (see Section 3.6). You should then compute $\mathbb{E}_q[\log \phi_{kw}]$ based on this initialized λ_{kw}. Then for T iterations, you will alternate between the E and M steps of the expectation-maximization algorithm to iteratively update γ and λ.

E-step:

1. Initialize variational parameter γ (see Section 3.6)
2. For each document $d = 1 \dots D$
   (a) For each iteration $m = 1 \dots M$
      i. Compute $\mathbb{E}_q[\log \theta_{dk}]$ based on most recent γ_{dk}
      $\mathbb{E}_q[\log \theta_{dk}] \leftarrow \Psi(\gamma_{dk}) - \Psi\left(\sum_{i=1}^{K} \gamma_{di}\right)$
      ii. Compute $\tilde{\pi}_{dik}$ based on current document d:
      $\tilde{\pi}_{dik} \propto \exp\left(\mathbb{E}_q[\log \theta_{dk}] + \mathbb{E}_q[\log \phi_{kx_{di}}]\right)$
      iii. Update variational parameter γ_d (for document-topic portions θ_d)
      $\gamma_{dk} \leftarrow \alpha + \sum_{i=1}^{N_d} \tilde{\pi}_{dik} x_{di}$
      iv. If $\frac{1}{K} \sum_k |\Delta \gamma_{dk}| < \epsilon$ break from loop early (change in γ_{dk} is smaller than ϵ).
3. Compute sufficient statistics $S(\lambda)$
   $S(\lambda) \leftarrow \sum_{d=1}^{D} \sum_{i=1}^{N_d} \tilde{\pi}_{dik} \cdot x_{di}^T$

M-step:

1. Update variational parameter λ_k (for topic-word distribution ϕ_k)
   $\lambda_{kv} \leftarrow \beta + S(\lambda)$

2. Compute $\mathbb{E}_q[\log \phi_{kw}]$ based on most recent λ_{kw}
   $\mathbb{E}_q[\log \phi_{kw}] \leftarrow \Psi(\lambda_{kw}) - \Psi\left(\sum_{v=1}^{V} \lambda_{kv}\right)$

You may use `scipy.special.digamma` to compute each $\Psi(\cdot)$ value.

Finally, you can approximate your parameters θ and ϕ once all EM iterations are complete, using your updated free variational parameters:

$$\theta = \gamma / \sum_{d=1}^{D} \gamma_d \tag{14}$$

$$\phi = \lambda / \sum_{v=1}^{V} \lambda_v \tag{15}$$

Important: If you are observing the same repeated words for every topic, chances are you are not normalizing γ and λ correctly. You must normalize the $\tilde{\pi}$ over all the topics k, that is, $\sum_{k=1}^{K} \tilde{\pi}_{dik}$

Tip for normalizing $\tilde{\pi}$:
You may find it helpful to split $\tilde{\pi}$ as

$$\exp\left(\mathbb{E}_q[\log \theta_{dk}]\right) \times \exp\left(\mathbb{E}_q[\log \phi_{kx_{di}}]\right)$$

and multiply by the data x_{di} as well as normalization constant in between the two terms. This will allow you to compute $\tilde{\pi}_{dik}$ for the just the document d being used, which would otherwise not have conformable dimensions.

Getting this to work out to the expected dimensions in the table is a little tricky, but is necessary for a functioning algorithm. Since γ_{dk} is indexed by d, the result of step 2.a.iii should be a vector of dimension K.

Table 1 summarizes notation used for your convenience.

3.6 Initialization

You will initialize your variational parameter λ at the start of fitting using a random gamma distribution. This has been implemented in the `initialize_lambda` method. We will be using the random seed 0 to ensure consistency across runs.

You will initialize your variational parameter γ at the start of each E-step using a random gamma distribution. This has been implemented in the `initialize_gamma` method. We will be using the random seed 0 to ensure consistency across runs.

3.7 Existing Components

The foundations of the learning framework have been provided for you. You will need to complete this library by filling in code where you see a `TODO` comment. **You should only make changes to** `models.py`. **Do not change the existing command line flags, existing filenames, or algorithm names.** We use these command lines to test your code. If you change their behavior, we cannot test your code.

Variable	**Description**	**Dimensions**
$1 \leq d \leq D$	Document index	
$1 \leq i \leq N_d$	Word index in document d	
$1 \leq v \leq V$	Word index in vocabulary	
$1 \leq k \leq K$	Topic index	
$1 \leq t \leq T$	Iteration timestep	
$1 \leq m \leq M$	Max iterations allowed for E-step convergence	
θ	Document-specific topic portion	$D \times K$
ϕ	Topic-word distribution	$K \times V$
z	Topic assignment to word w in document d	$D \times V$
x_{di}	Frequency of word i in document d	N_d
λ	Variational parameter for ϕ_k	$K \times V$
γ	Variational parameter for θ_d	$D \times K$
π	Variational parameter for z_{di}	$D \times V \times K$
$\mathbb{E}_q[\log \theta_{dk}]$	Expectation under q of $\log \theta$	K
$\mathbb{E}_q[\log \phi_{kw}]$	Expectation under q of $\log \phi$	$K \times V$
$\mathbb{E}_q[\log \phi_{kx_{di}}]$	Expectation under q of $\log \phi$ for words in d	$K \times N_d$
$S(\lambda)$	Sufficient statistic for λ implicitly computing π for each document d	$K \times V$
ϵ	Threshold for convergence in E-step	scalar
α, β	Dirichlet hyperparameters	scalar

Table 1: Summary of notation. Includes variable descriptions and expected dimensions.

The code we have provided is fairly compact, and you should spend some time to familiarize yourself with it. Here is a short summary to get you started:

- `data.py` –This contains the `build_dtm` function, reads in all the documents from the specified directory and returns the document-word matrix and vocab list. The document-word matrix is stored as a sparse matrix of 32-bit integers (and in particular as a `scipy.sparse.coo_matrix` of `np.intc`), which has `num_docs` rows and `|vocab|` columns. The index of words in the vocab list correspond to their column indices in the document-term matrix.

- `main.py` – This is the main testbed to be run from the command line. It takes care of parsing command line arguments, entering train/test mode, saving models/predictions, etc. You should not need to make any changes in this file this time.

- `models.py` – This contains a `Model` class which you should extend. We have implemented the LDA class for you, which calls the inference methods you will write and infers the words associated with each topic in the `predict` method. There is also an Inference class which is extended into the MeanFieldVariationalInference class you will implement. At the very least, you should have `initialize` and `inference` methods, but you are encouraged to add other methods as necessary.

- `compute_topic_coherence.py` – This is a script which measures how well your inference did in uncovering the topic-word distribution. We're using Normalized Pointwise Mutual Information (NPMI)[12] as a measure of the topic coherence. NPMI

[12]Bouma, G. (2009). Normalized (pointwise) mutual information in collocation extraction.

ranges from -1 to 1, with -1 indicating that the pair of tags never occurred together, 0 indicating that the tag occurrences were independent of each other, and 1 indicating that the tags co-occurred perfectly with each other.

- `run.sh` – For your convenience, we have included a bash script that runs all the commands (no more needing to write all those command arguments!). If you open the bash script in your text editor, you can specify the data directory and hyperparameters that are passed to the respective commands. You can run the script as follows:

```
bash run.sh
```

Your code must correctly implement the behavior described by the following commandline options, which are passed in as arguments to the constructor of your implementation in models.py.

- `--data` This is required! Specifies the directory where the documents in the corpus are located.
- `--predictions-file` This is required! Tells the program where to store a model's topic assignments over words in the vocab.
- `--num-documents` Specifies the number of documents from the corpus to include within the document-word matrix. By default, all documents are included.
- `--max-words` Specifies the number of words to include (after pre-processing) from each document. By default, 1000 words are kept.
- `--top-k` Specifies the the top k words in each topic to include in predictions. By default, this is set to 10.

In addition to these meta arguments, `main` also takes some hyperparameter arguments, namely:

- `--num-topics` The number of topics to uncover in the data. By default, this is set to 10.
- `--alpha` The Dirichlet parameter for document-portions (θ_d). By default this parameter is set to 0.1.
- `--beta` The Dirichlet parameter for topic-word distribution (ϕ_k). By default this parameter is set to 0.01.
- `--epsilon` The threshold for E-step convergence. By default this parameter is set to 0.001.
- `--num-vi-iterations` Model hyper-parameter controlling the number of iterations to use during Variational Inference (T). By default, this value is 100.
- `--num-estep-iterations` Model hyper-parameter controlling the number of iterations to use during the E-step (M). This is just defined to prevent infinite loops in the case of implementation bugs. By default, this value is 50.

3.7.1 Running Inference

The usage for this framework is:

```
python3 main.py --predictions-file predictions_file --data data_directory
```

Each assignment will specify the string argument for an algorithm. The `data` option indicates the document directory. Finally, results are saved to the `predictions-file`.

3.7.2 Examples

This runs inference and prints the topics for speeches delivered from 1790-2020:

```
python3 main.py --data speeches --predictions-file predictions
```

4 Tips and Tricks

You may find this section helpful when working with the data structures and tracking your algorithm progress.

4.1 Working with `coo_matrix`

The document-term matrix created in `data.py` is packaged into a `coo_matrix`[13], which is a sparse COOrdinate format[14]. `coo_matrix` are used to quickly construct sparse matrices and contain a few useful attributes for this problem. Specifically, the attributes you may find useful are:

- `nnz`: returns number of stored values
- `data`: returns stored values in COO format
- `row`: returns row index of stored values in COO format
- `col`: returns column index of stored values in COO format

Let's consider the simple matrix to see these attributes in practice:

```
>>> A = np.array([[0, 3, 1], [4, 0, 2], [0, 0, 7]])
>>> A
array([[0, 3, 1],
       [4, 0, 2],
       [0, 0, 7]])
A = sparse.coo_matrix(A, dtype=np.intc)
>>> A
<3x3 sparse matrix of type '<class 'numpy.int32'>'
with 5 stored elements in COOrdinate format>
>>> A.nnz
5
>>> A.data
array([3, 1, 4, 2, 7], dtype=int32)
>>> A.row
```

[13] Documentation: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html`

[14] Also known as 'ijv' format: `http://scipy-lectures.org/advanced/scipy_sparse/coo_matrix.html`

```
array([0, 0, 1, 1, 2], dtype=int32)
>>> A.col
array([1, 2, 0, 2, 2], dtype=int32)
```

In this homework, you will be working with a document-word matrix, where the row index represents the document in the corpus and the column index represents the word in the vocab. The above attributes make iterating over documents and words in the document-word matrix seamless (hint: they all return the same length vector!).

4.2 Help! My models are taking forever to run!

If you find your models are running *very* slowly, try decreasing the number of documents from the corpus included within the document-word matrix or the number of words included from each document. You can adjust the number of SotU addresses included by using the command line arg `--num-documents`. By default, all addresses are used. You may want to decrease this further while you debug your implementation.

4.3 Track Inference Progress with `tqdm`

`tqdm`[15] has been included in the `requirements.txt` file. You can use it to add a smart progress bar to your code that informs you which iteration you're on, the elpased time and estimated time left, and finally time required per iteration. To use it, just wrap an iterable (e.g. `range(iterations)`) in it as such `tqdm(iterable)`. An example progress bar is shown below:

```
13%|||||||                                  | 128/1000 [02:25<15:18,  1.05s/it]
```

Important: Before submitting your code to Gradescope, make sure to remove `tqdm` from your for loops. This will mess up the autograder.

5 Miscellaneous

5.1 Python Libraries

We will be using Python 3.7.6. We are *not* using Python 2, and will not accept assignments written for this version. We recommend using a recent release of Python 3.7.x, but anything in this line (e.g. 3.x) should be fine.

For each assignment, we will tell you which Python libraries you may use. We will do this by providing a `requirements.txt` file. You should only use libraries available in the basic Python library or the `requirements.txt` file. In this and future assignments we will allow you to use `numpy`, `scipy`, and `tqdm`. Note, `nltk` is included in the `requirements.txt` for text processing in `data.py`. However, you may *not* use functions from this library.
It may happen that you find yourself in need of a library not included in the `requirements.txt` for the assignment. You may request that a library be added by posting to Piazza. This may be useful when there is some helpful functionality in another library that we omitted. However, we are unlikely to include a library if it either solves a major part of the assignment, or includes functionality that isn't really necessary.

[15] Documentation: `https://github.com/tqdm/tqdm`

5.2 Grading Programming

The programming section of your assignment will be graded using an autograder in Gradescope. Your code will be run using the provided command line options, as well as other variations on these options (different parameters, data sets, etc.) The grader will consider the following aspects of your code.

1. **Exceptions:** Does your code run without crashing?
2. **Output:** Some assignments will ask you to write some data to the console. Make sure you follow the provided output instructions exactly.
3. **Accuracy:** If your code works correctly, then it should achieve a certain accuracy on each data set. While there are small difference that can arise, a correctly working implementation will get the right answer.
4. **Speed/Memory:** As far as grading is concerned, efficiency largely doesn't matter, except where lack of efficiency severely slows your code (so slow that we assume it is broken) or the lack of efficiency demonstrates a lack of understanding of the algorithm. For example, if your code runs in two minutes and everyone else runs in 2 seconds, you'll lose points. Alternatively, if you require 2 gigs of memory, and everyone else needs 10 MB, you'll lose points. In general, this happens not because you did not optimize your code, but when you've implemented something incorrectly.

5.3 Code Readability and Style

In general, you will not be graded for code style. However, your code should be readable, which means minimal comments and clear naming / organization. If your code works perfectly then you will get full credit. However, if it does not we will look at your code to determine how to allocate partial credit. If we cannot read your code or understand it, then it is very difficult to assign partial credit. Therefore, it is in your own interests to make sure that your code is reasonably readable and clear.

5.4 Code Structure

Your code must support the command line options and the example commands listed in the assignment. Aside from this, you are free to change the internal structure of the code, write new classes, change methods, add exception handling, etc. However, once again, do not change the names of the files or command-line arguments that have been provided. We suggest you remember the need for clarity in your code organization.

5.5 Knowing Your Code Works

How do you know your code really works? That is a very difficult problem to solve. Here are a few tips:

1. Use Piazza. While **you cannot share code**, you can share results. We encourage you to post your results on dev data for your different algorithms. A common result will quickly emerge that you can measure against.
2. Output intermediate steps. Looking at final predictions that are wrong tells you little. Instead, print output as you go and check it to make sure it looks right. This can also be helpful when sharing information on Piazza.

3. Debug. Find a Python debugger that you like and use it. This can be very helpful.

5.6 Debugging

The most common question we receive is "how do I debug my code?" The truth is that machine learning algorithms are very hard to debug because the behavior of the algorithm is unknown. In these assignments, you won't know ahead of time what accuracy is expected for your algorithm on a dataset. This is the reality of machine learning development, though in this class you have the advantage of your classmates, who may post the output of their code to the bulletin board. While debugging machine learning code is therefore harder, the same principles of debugging apply. Write tests for different parts of your code to make sure it works as expected. Test it out on the easy datasets to verify it works and, when it doesn't, debug those datasets carefully. Work out on paper the correct answer and make sure your code matches. Don't be afraid of writing your own data for specific algorithms as needed to test out different methods. This process is part of learning machine learning algorithms and a reality of developing machine learning software.

6 What to Submit

You will need to create an account on gradescope.com and signup for this class. The course is `https://www.gradescope.com/courses/153788`.

Submit your code (.py files) to gradescope.com as a zip file. **Your code must be uploaded as code.zip with your code in the root directory**. By 'in the root directory,' we mean that the zip should contain `*.py` at the root (`./*.py`) and not in any sort of substructure (for example `hw6/*.py`). One simple way to achieve this is to zip using the command line, where you include files directly (e.g., `*.py`) rather than specifying a folder (e.g., `hw6`).

7 Questions?

Remember to submit questions about the assignment to the appropriate group on Piazza: `https://piazza.com/class/kdfbbwz3hwb3an`.