CS 475 Machine Learning: Homework 5
Graphical Models, Inference, and Structured Prediction
Programming Assignment

Due: Saturday May 2, 2020, 11:59pm

60 Points Total Version 1.0

Before you begin, please read the Homework 5 Introduction. The introduction has information on how to use the Python codebase, the data, and how to submit your assignment. Make sure to read this pdf in it's entirety before beginning your implementations. There are some helpful tips and tricks in section 4 of this document.

We have provided Python code to serve as a testbed for your algorithms. We'll use the same coding framework that we used in the first two homework 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` file; do not modify any of the other files.

In this assignment you will be exploring a technique for natural language understanding known as Topic Modeling. You will be implementing sampling and variational inference methods for learning the parameters of the probabilistic LDA graphical model.

1 Background

1.1 Topic Model

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[1]. 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.

1.2 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)[2]. In the LDA model, documents are represented as random mixtures over latent topics, whereas topics are characterized as distributions over words. For each document d in the corpus, the generative story of LDA is as follows:

[1] 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.

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

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 $z_i \sim \text{Multinomial}(\theta_d)$
      ii. Choose a word $w_i \sim \text{Multinomial}(\phi_{z_i})$

where α and β are hyperparameters of the two Dirichlet distributions. These are typically small values chosen to be < 1. Given these two parameters, the joint distribution of the topic model is expressed:

$$P(w, z, \theta, \phi|\alpha, \beta) = \prod_{i=1}^{K} P(\phi_k|\beta) \prod_{d=1}^{D} P(\theta_d|\alpha) \prod_{t=1}^{N_d} P(z_t|\theta_d) P(W_t|\phi_{z_t}) \tag{1}$$

This generative story is concisely represented using plate notation:

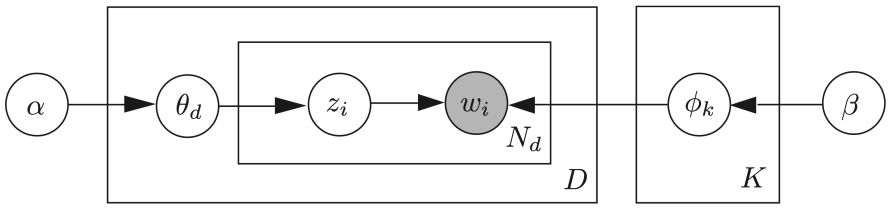

Figure 1: Three-layer hierarchal Bayesian LDA graphical model. Here, the use of plate notation represents replication. That is, document d is repeated D times in the corpus, the word tokens w_n are repeated N_d times in each document d. There is a fixed vocabulary size of W and K topics. The document-specific topic proportion $\theta_d(k)$ generates a topic label $z^k_{w,d} \in \{0, 1\}$ such that $\sum_{k=1}^{K} z^k_{w,d} = 1$. These topic labels $z_{w,d}$ then generate each observed word token i at index w in the document d based on the topic-specific multinomial distribution $\phi_k(w)$ over all words in the vocabulary. The multinomial distributions $\theta_d(k)$ and $\phi_k(w)$ 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, expressed through their topic distribution ϕ. θ_d is a low-dimensional representation of document d in topic space, where z_i represents which topics generated a particular word w_i. To leverage this power, we have to first learn the posterior distributions of the latent variables in the topic model given the observed documents:

$$P(\theta, \phi, z|w, \alpha, \beta) = \frac{P(\theta, \phi, z, w|\alpha, \beta)}{P(w|\alpha, \beta)} \tag{2}$$

However, $P(w|\alpha, \beta)$ cannot be computed exactly and so computing the posterior becomes intractable. Thus, we can make use of approximate inference methods to estimate this posterior distribution. In particular, you will be implementing Gibbs Sampling and Belief Propagation (Sum-Product).

2 Collapsed Gibbs Sampling (25 points)

Gibbs Sampling belongs to the family of Markov Chain Monte Carlo (MCMC) algorithms[3]. The goal of MCMC algorithms is to construct a Markov chain whose stationary distribution is the target posterior distribution. That is, after stepping through this Markov chain for a number of iterations, the points sampled from this constructed distribution should converge to points if sampled from the true posterior. Gibbs Sampling is a variant that samples from *conditional distributions* of the variables of the posterior.

2.1 Inference

Your `GibbsSampling` class should inherit from `Inference`. In the topic model paradigm there are three latent variables that can be uncovered using inference. These include: the latent document-topic portions θ_d, the topic-word distributions ϕ_{z_i}, and the topic index assignments for each word z_i. Each of these three latent variables can be modeled with a conditional probability distribution. However, we can exploit the fact that $\mathbf{z}$ is a sufficient statistic for θ_d and ϕ_{z_i} to simplify our inference problem. **Specifically, the multinomial parameters θ_d and ϕ_{z_i} can be integrated out of LDA, allowing all sampling to be done from the conditional distribution for $\mathbf{z}$**[4]. The multinomial paramaters θ_d and ϕ_{z_i} can then be computed using $\mathbf{z}$:

$$\theta_{d,z} = \frac{n_{d,z} + \alpha}{\sum_{z=1}^{Z} n_{d,z} + \alpha}, \qquad \phi_{z,w} = \frac{n_{z,w} + \beta}{\sum_{w=1}^{W} n_{z,w} + \beta} \tag{3}$$

This modification—integrating out the multinomial parameters and only sampling from a simple discrete distribution with easily computable parameters—is known as *Collapsed* Gibbs Sampling.

Your implementation of the posterior update equation will maintain a few count variables (note we use the latent variable z to represent topic assignments instead of k which are the "true" topics in the data):

- $n_{d,z}$: the number of words assigned to topic z in document d
- $n_{z,w}$: the number of times word w is assigned topic z
- n_z: the number of times any word is assigned to topic z

You can think of these count variables as *unnormalized* probabilities. So maintaining these count variables updates the conditional probability tables of the directed graphical model.

The objective of the collapsed Gibbs sampler for LDA is to estimate the topic index posterior (i.e. topic z_i assigned to word w_i) using the previous topic assignments on all the words:

$$p(z_i | z^{(-i)}, \alpha, \beta, \mathbf{w}) \tag{4}$$

[3] W. M. Darling, "A Theoretical and Practical Implementation Tutorial on Topic Modeling and Gibbs Sampling," in Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, 2011, pp. 642–647.

[4] For an in-depth mathematical treatment, see Integrating Out Multinomial Parameters in Latent Dirichlet Allocation and Naive Bayes for Collapsed Gibbs Sampling

where $z^{(-i)}$ represents all topic assignments except for z_i. Using the chain rule, the Gibbs sampler (i.e. the posterior distribution of the current topic assignment given the word and previous topic assignments) for the topic index z_i is:

$$p(z_i|\mathbf{z}^{(-i)}, \mathbf{w}, \alpha, \beta) = \frac{p(\mathbf{w}, \mathbf{z}|\alpha, \beta)}{p(\mathbf{w}, \mathbf{z}^{(-i)}|\alpha, \beta)} = \frac{p(\mathbf{z}|\alpha, \beta)}{p(z^{(-i)}|\alpha, \beta)} \cdot \frac{p(\mathbf{w}|\mathbf{z}, \alpha, \beta)}{p(\mathbf{w}^{(-i)}|\mathbf{z}^{(-i)}, \alpha, \beta)p(w_i)} \tag{5}$$

$$\propto \frac{(n_{d,z}^{(-i)} + \alpha) \times (n_{z,w}^{(-i)} + \beta)}{\sum_{j=1}^{|W|} n_{z,w_j}^{(-i)} + \beta_j} \tag{6}$$

The posterior update equation in Eq. 6 requires you to sample from a distribution conditioned on every topic assignment thus far except the current one. So you will have to first remove the current topic assignment from the equation. This is accomplished by *decrementing* every count variable associated with the current topic assignment.Why does this work? It is because LDAs are *exchangeable* (see section 2.6 for more details).

Eq. 6 is the conditional distribution used to construct the Markov chain. You will sample from this Markov chain and use this posterior distribution to determine the current topic assignment[5]. **Choose the topic assignment with the maximum probability**. Finally, update all count variables. And that's it, after a specified number of iterations, these topic assignments sampled from the Markov chain will converge to points sampled from the true, unknown topic distribution.

After your model completes all iterations, the count variables will be used to compute the latent distributions θ_d and ϕ_k.

2.2 Implementation Details

The number of iterations is specified by the command arg `--iterations` and has a default value of 100. The Dirichlet parameters have default values 0.1 and 0.01 for α and β, respectively. The Collapsed Gibbs Sampling procedure for LDA is summarized:

- For each word w in each document d
  - topic $\leftarrow z_{w,d}$
  - Decrement counters $n_{d,z}$, $n_{z,w}$, n_z by one
  - Compute posterior $p(z = k|\cdot) = \frac{(n_{d,z}^{(-i)}+\alpha)\times(n_{z,w}^{(-i)}+\beta)}{\sum_{j=1}^{|W|} n_{z,w_j}^{(-i)}+\beta_j}$
  - topic $\leftarrow$ sample from posterior $p(z|\cdot)$
  - Increment counters $n_{d,z}$, $n_{z,w}$, n_z by one
- Compute model log-likelihood by calling `self._loglikelihood()` and append to `self.loglikelihoods`
- Repeat above for t iterations
- Compute $\theta = \frac{n_{dz}+\alpha}{\sum_{z=1}^{Z} n_{dz}+\alpha}$ and $\phi = \frac{n_{zw}+\beta}{\sum_{w=1}^{W} n_{zw}+\beta}$

[5]Hint: `numpy` has a nice function for sampling from multinomial probabilities. You can find the documentation here: `https://docs.scipy.org/doc/numpy-1.14.1/reference/generated/numpy.random.multinomial.html`

2.3 Initialization

You will need to initialize z (your data structure that contains topic assignments for each word in each document) before beginning the inference algorithm. Typically in Gibbs Sampling for LDA, you would initialize z by randomly assigning a topic to each word in each document and then incrementing the appropriate counts.

Like usual, for the sake of ensuring consistent results among everyone, you will use the following initialization scheme:

- Start a counter $t = 0$
- For each word w in each document d
  - Assign w topic t mod K and save in $z_{w,d}$ (where K is the number of topics)
  - Increment counters $n_{d,z}$, $n_{z,w}$, n_z by one
  - Increment t

(Hint: you will want to use a sparse data structure for z such as a dictionary or a sparse matrix).

2.4 Convergence

While Gibbs Sampling does guarantee convergence to what would be sampled from the true posterior distribution, it is usually difficult to determine how many iterations are required to reach the stationary distribution. In practice, we can use the log-likelihood to estimate convergence of our inference. We have provided a plot in the jupyter notebook `visualizing_results.ipynb` where you can visualize the negative log likelihood to see if your model has converged.

2.5 Burning in the Gibbs Sampler

Burn-in gives the Markov Chain time to reach its equilibrium distribution. The core idea of burn-in is to discard the first few iterations of an MCMC run, before sampled points are saved. The intuition behind this practice is that it prevents over-representation of rare points that may saturate your samples if the Markov Chain started at a bad initialization point. **You don't have to implement anything for this part**, but when you view your log-likelihood plots in the notebook, observe the rate of change during the first few iterations as the Gibbs sampler is burning in.

2.6 Exchangeability

A finite set of random variables $\{z_1, ..., z_N\}$ is said to be *exchangeable* if the joint distribution is invariant to permutation. Suppose π is a permutation of the integers from 1 to N. Then,

$$p(z_1, ..., z_N) = p(z_{\pi(1)}, ..., z_{\pi(N)}) \tag{7}$$

Furthermore, an infinite sequence of random variables is said to be *infinitely exchangeable* if every finite subsequence is exchangeable. The assumption in LDA is that words are generated from topics that are infinitely exchangeable within a document.

3 The Sum Product Algorithm (25 points)

Belief Propagation is a message-passing algorithm that passes real-valued functions between hidden (latent) nodes in a graphical model across its edges. The messages passed between nodes in the network specify the *influence* one variable exerts over another. The beliefs give the posterior probability after certain events are observed. At a particular node, the beliefs approximate the marginal posterior probability by multiplying incoming messages with local evidence. The goal of belief propagation is to estimate the marginal distribution $\prod_i p_i(x_i)$ of a graphical model.

When expressed over factor graphs, belief propagation becomes the Sum-Product algorithm. A factor graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ is a bipartite graph between variable and factor nodes, with the joint probability distribution:

$$p(x_1, \cdots, x_N) = \frac{1}{Z} \prod_{j=1}^{m} f_j(x_{f_j})$$

where x_{f_j} is the vector of neighboring variable nodes to factor f_j. Factor graphs capture additional structure that both directed and undirected graphical models do not.

3.1 Inference

You will implement Belief Propagation using the Sum-Product algorithm. Figure 2 shows an equivalent factor graph representation of the three-layer hierarchal Bayesian LDA graphical model shown in Figure 1.

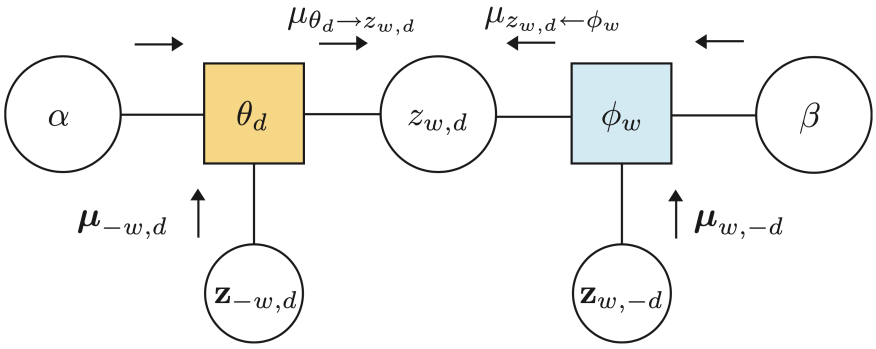

Figure 2: Factor Graph of LDA with message passing.

From an MRF perspective, the objective of the topic model becomes assigning a set of semantic topic labels $\mathbf{z} = \{z^k_{w,d}\}$ to explain nonzero elements within the document-word matrix $\mathbf{x} = \{x_{w,d}\}$. Topic assignments are made using the MAP (*maximum a posteriori*, i.e. maximum posterior) estimate through maximizing the posterior probability[6].

For reference, the following messages will be used in your implementation of the Sum-Product algorithm:

- $\mu_{w,d}(k)$: Conditional marginal probability of topic z being assigned to word w in document d. Proportional to product of incoming messages from factors θ_d and ϕ_w.

[6] J. Zeng, W. K. Cheung and J. Liu, "Learning Topic Models by Belief Propagation," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 35, no. 5, pp. 1121-1134, May 2013.

- $\mu_{\theta_d \to z_{w,d}}(k)$: Message sent from factor node θ_d to the variable $z_{w,d}$. Proportional to sum of incoming messages from neighboring variables $\mu_{-w,d}(k)$.
- $\mu_{\phi_w \to z_{w,d}}(k)$: Message sent from factor node ϕ_w to the variable $z_{w,d}$. Proportional to sum of incoming messages from neighboring variables $\mu_{w,-d}(k)$.

In the LDA factor graph, messages $\mu_{w,d}(k)$ represent the (unnormalized) conditional marginal probabilities, $p(z^k_{w,d} = 1, x_{w,d} | \mathbf{z}^k_{-w,-d}, \mathbf{x}_{-w,-d})$. Using the Markov property, these messages can be expressed:

$$p(z^k_{w,d}, x_{w,d} | \mathbf{z}^k_{-w,-d}, \mathbf{x}_{-w,-d}) \propto p(z^k_{w,d}, x_{w,d} | \mathbf{z}^k_{-w,d}, \mathbf{x}_{-w,d}) p(z^k_{w,d}, x_{w,d} | \mathbf{z}^k_{w,-d}, \mathbf{x}_{w,-d}) \tag{8}$$

$$\mu_{w,d}(k) \propto \frac{\sum_{-w} x_{-w,d} z^k_{-w,d} + \alpha}{\sum_{k=1}^{K} (\sum_{-w} x_{-w,d} z^k_{-w,d} + \alpha)} \times \frac{\sum_{-d} x_{w,-d} z^k_{w,-d} + \beta}{\sum_{w=1}^{W} (\sum_{-d} x_{w,-d} z^k_{w,-d} + \beta)} \tag{9}$$

where $-w$ and $-d$ denote all word indices except w and all document indices except d, and the notations $\mathbf{z}_{-w,d}$ and $\mathbf{z}_{w,-d}$ represent all possible neighboring topic configurations. In Eq. 9, message updates use the variable $z^k_{w,d}$ which depends on its neighboring topic configuration $\{\mathbf{z}^k_{-w,d}, \mathbf{z}^k_{w,-d}\}$. Since the true topic configurations are not known, these configurations are replaced by messages from the factor nodes to the variable node $z^k_{w,d}$:

$$\mu_{w,d}(k) \propto \mu_{\theta_d \to z_{w,d}}(k) \times \mu_{\phi_w \to z_{w,d}}(k) \tag{10}$$

where $\mu_{\theta_d \to z_{w,d}}(k)$ represents the message sent from factor node θ_d to the variable $z_{w,d}$ and $\mu_{\phi_w \to z_{w,d}}(k)$ represents the message from factor node ϕ_w to the variable $z_{w,d}$. Individually, these messages are:

$$\mu_{\theta_d \to z_{w,d}}(k) = \frac{\boldsymbol{\mu}_{-w,d}(k) + \alpha}{\sum_k [\boldsymbol{\mu}_{-w,d}(k) + \alpha]} \text{ where } \boldsymbol{\mu}_{-w,d}(k) = \sum_{-w} x_{-w,d} \mu_{-w,d}(k) \tag{11}$$

$$\mu_{\phi_w \to z_{w,d}}(k) = \frac{\boldsymbol{\mu}_{w,-d}(k) + \beta}{\sum_w [\boldsymbol{\mu}_{w,-d}(k) + \beta]} \text{ where } \boldsymbol{\mu}_{w,-d}(k) = \sum_{-d} x_{w,-d} \mu_{w,-d}(k) \tag{12}$$

Observe that the messages are weighted during propagation using the word counts $x_{w,d}$. This means that larger $x_{w,d}$ have a greater influence of the message on its neighbors. See section 3.4 for why this can potentially be problematic.

Messages are passed from variables to factors, and in turn from factors to variables until convergence or the maximum number of iterations is reached.

To estimate parameters θ_d and ϕ_w, we use the messages $\mu_{w,d}$:

$$\theta_d(k) = \frac{\boldsymbol{\mu}_{\cdot,d}(k) + \alpha}{\sum_k [\boldsymbol{\mu}_{\cdot,d}(k) + \alpha]} \text{ where } \boldsymbol{\mu}_{\cdot,d}(k) = \sum_{w} x_{w,d} \mu_{w,d}(k) \tag{13}$$

$$\phi_w(k) = \frac{\boldsymbol{\mu}_{w,\cdot}(k) + \beta}{\sum_w [\boldsymbol{\mu}_{w,\cdot}(k) + \beta]} \text{ where } \boldsymbol{\mu}_{w,\cdot}(k) = \sum_{d} x_{w,d} \mu_{w,d}(k) \tag{14}$$

3.2 Implementation Details

You should only pass messages when $x_{w,d} \neq 0$ (i.e. word w exists in document d). We *strongly* recommend that you work with X (the document-word matrix) as a sparse `coo_matrix` instead of converting it to a dense matrix for this problem. The document-word matrix will be large and very sparse, so iterating over each element will be computationally inefficient. `coo_matrix` has a few handy methods for getting non-zero elements within the data structure. See section 4.1 for more details.

The Sum-Product message-passing procedure for LDA is summarized:

- For each word w in each document d
  - Sum incoming messages from variable node $z_{-w,d}$ to factor node θ_d
  - Sum incoming messages from variable node $z_{w,-d}$ to factor node ϕ_k
  - Multiply incoming messages from factor nodes θ_d and ϕ_k to variable node $z_{w,d}$
  - Update neighboring messages, $\mu_{-w,d}$ and $\mu_{w,-d}$, based on current $\mu_{w,d}$ (hint: look how this is done during initialization in section 3.3)
- Compute model loglikelihood by calling `self._loglikelihood(X)` and append to `self.loglikelihoods`
- Repeat above for t iterations
- Compute $\theta = \frac{\boldsymbol{\mu}_{\cdot,d}(k)+\alpha}{\sum_k[\boldsymbol{\mu}_{\cdot,d}(k)+\alpha]}$ and $\phi = \frac{\boldsymbol{\mu}_{w,\cdot}(k)+\beta}{\sum_w[\boldsymbol{\mu}_{w,\cdot}(k)+\beta]}$

Based on the dimensions of the $\mu_{w,d}$, in Eq. 14 you will actually be calculating ϕ_k^T. Make sure to transpose this when setting `self.phi` as this is what the `predict` method expects.

Hint: it may be easier to multiply the data when updating $\mu_{-w,d}$ and $\mu_{w,-d}$ so that you don't have to do so when updating Equations (13) and (14).

Belief Propagation can be implemented both synchronously (by waiting for all incoming messages to arrive before propagating) and asynchronously (sending messages as soon as they arrive). Asynchronous implementations have been shown to converge faster. **For this assignment, you should implement the synchronous version**.

3.3 Initialization

You will need to initialize your messages from factor to variable node, $\mu_{w,d}$, $\mu_{-w,d}$, $\mu_{w,-d}$, with topic assignments before beginning message-passing. You can do this by assigning random topics to each word in each document within $\mu_{w,d}$ and then weighting these messages to initialize $\mu_{-w,d}$ and $\mu_{w,-d}$. For the sake of more consistent results, we will deterministically be assigning topic assignments in $\mu_{w,d}$. This is summarized in the following initialization scheme:

- Randomly assign topics for each word in each document in $\mu_{w,d}$ and normalize (**this has been done for you**)
- For each word w in each document d
  - $\mu_{-w,d} \leftarrow x_{w,d} \times \mu_{w,d}(k)$
  - $\mu_{w,-d} \leftarrow x_{w,d} \times \mu_{w,d}(k)$

3.4 Message Weighting

Message weighting is relevant because, as we saw in Equations (11) and (12), documents with higher word counts will have a larger influence on topic assignments, which may not be desirable. In practice, we can use more sophisticated term weighting functions like term-frequency (TF) or term-frequency inverse-document-frequency (TF-IDF) to make this more robust. We will not be asking you to implement these techniques for this assignment.

4 Tips and Tricks

These are relevant for both problems in the assignment.

4.1 Working with `coo_matrix`

The document-term matrix created in `data.py` is packaged into a `coo_matrix`[7], which is a sparse COOrdinate format[8]. `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
array([0, 0, 1, 1, 2], dtype=int32)
>>> A.col
array([1, 2, 0, 2, 2], dtype=int32)
```

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

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

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. You can adjust the number of documents from a particular blogger's corpus to include using the command line arg `--num-documents`. By default, 100 documents from the corpus are included. You may want to decrease this further while you debug your implementations.

4.3 Track Inference Progress with `tqdm`

`tqdm`[9] 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 Notebook (10 points)

A rule of thumb, when beginning a new project, is to ask yourself three questions:

1. What is the problem I'm trying to solve?
2. Why is this problem important?
3. How will I know if I've succeeded?

The third question is challenging to answer when using Unsupervised Learning. There are no ground truth labels. The learned clusters aren't "right" or "wrong". Unsupervised Learning is successful if the learned topics are useful: if they give insight into the data. You will evaluate the performance of your topic models in the provided notebook `visualizing_results.ipynb`.

In this notebook, we have provided code for you to visualize the topics your models uncover. You do **not** need to add/modify any code for the visualizations, but we will use the graph outputs to grade your code. **There are questions in this notebook you will need to answer**. You will write out your answers to the questions in the notebook, save your notebook as a PDF, and upload the PDF to Gradescope. Remember to run all the cells and make sure the output graphs from your code are visible in the PDF.

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

6 Submitting

Please refer to Homework 5 Introduction for instructions on submitting your programming assignment.