3 Gibbs Sampling and LDA

Lab Objective: *Understand the basic principles of implementing a Gibbs sampler. Apply this to Latent Dirichlet Allocation.*

Gibbs Sampling

Gibbs sampling is an MCMC sampling method in which we construct a Markov chain which is used to sample from a desired joint (conditional) distribution

$$\mathbb{P}(x_1, \cdots, x_n|\mathbf{y}).$$

Often it is difficult to sample from this high-dimensional joint distribution, while it may be easy to sample from the one-dimensional conditional distributions

$$\mathbb{P}(x_i|\mathbf{x}_{-i}, \mathbf{y})$$

where $\mathbf{x}_{-i} = x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n$.

Algorithm 3.1 Basic Gibbs Sampling Process.

```
procedure Gibbs Sampler
    Randomly initialize x_1, x_2, ..., x_n.
    for k = 1, 2, 3, ... do
        for i = 1, 2, ..., n do
            Draw x ~ P(x_i | x_{-i}, y)
            Fix x_i = x
        x^(k) = (x_1, x_2, ..., x_n)
```

A Gibbs sampler proceeds according to Algorithm 3.1. Each iteration of the outer for loop is a *sweep* of the Gibbs sampler, and the value of $\mathbf{x}^{(k)}$ after a sweep is a *sample*. This creates an irreducible, non-null recurrent, aperiodic Markov chain over the state space consisting of all possible $\mathbf{x}$. The unique invariant distribution for the chain is the desired joint distribution

$$\mathbb{P}(x_1, \cdots, x_n|\mathbf{y}).$$

Thus, after a burn-in period, our samples $\mathbf{x}^{(k)}$ are effectively samples from the desired distribution.

Consider the dataset of N scores from a calculus exam in the file `examscores.npy`. We believe that the spread of these exam scores can be modeled with a normal distribution of mean μ and variance σ^2. Because we are unsure of the true value of μ and σ^2, we take a Bayesian approach and place priors on each parameter to quantify this uncertainty:

$$\begin{aligned} \mu &\sim N(\nu, \tau^2) && \text{(a normal distribution)} \\ \sigma^2 &\sim IG(\alpha, \beta) && \text{(an inverse gamma distribution)} \end{aligned}$$

Letting $\mathbf{y} = (y_1, \dots, y_N)$ be the set of exam scores, we would like to update our beliefs of μ and σ^2 by sampling from the posterior distribution

$$\mathbb{P}(\mu, \sigma^2 | \mathbf{y}, \nu, \tau^2, \alpha, \beta).$$

Sampling directly can be difficult. However, we *can* easily sample from the following conditional distributions:

$$\begin{aligned} \mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2, \alpha, \beta) &= \mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \nu, \tau^2, \alpha, \beta) &= \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta) \end{aligned}$$

The reason for this is that these conditional distributions are *conjugate* to the prior distributions, and hence are part of the same distributional families as the priors. In particular, we have

$$\begin{aligned} \mathbb{P}(\mu | \sigma^2, \mathbf{y}, \nu, \tau^2) &= N(\mu^*, (\sigma^*)^2) \\ \mathbb{P}(\sigma^2 | \mu, \mathbf{y}, \alpha, \beta) &= IG(\alpha^*, \beta^*), \end{aligned}$$

where

$$\begin{aligned} (\sigma^*)^2 &= \left(\frac{1}{\tau^2} + \frac{N}{\sigma^2} \right)^{-1} \\ \mu^* &= (\sigma^*)^2 \left(\frac{\nu}{\tau^2} + \frac{1}{\sigma^2} \sum_{i=1}^{N} y_i \right) \\ \alpha^* &= \alpha + \frac{N}{2} \\ \beta^* &= \beta + \frac{1}{2} \sum_{i=1}^{N} (y_i - \mu)^2 \end{aligned}$$

We have thus set this up as a Gibbs sampling problem, where we have only to alternate between sampling μ and sampling σ^2. We can sample from a normal distribution and an inverse gamma distribution as follows:

```
>>> from math import sqrt
>>> from scipy.stats import norm
>>> from scipy.stats import invgamma
>>> mu = 0. # the mean
>>> sigma2 = 9. # the variance
>>> normal_sample = norm.rvs(mu, scale=sqrt(sigma))
>>> alpha = 2.
>>> beta = 15.
>>> invgamma_sample = invgamma.rvs(alpha, scale=beta)
```

Note that when sampling from the normal distribution, we need to set the `scale` parameter to the standard deviation, *not* the variance.

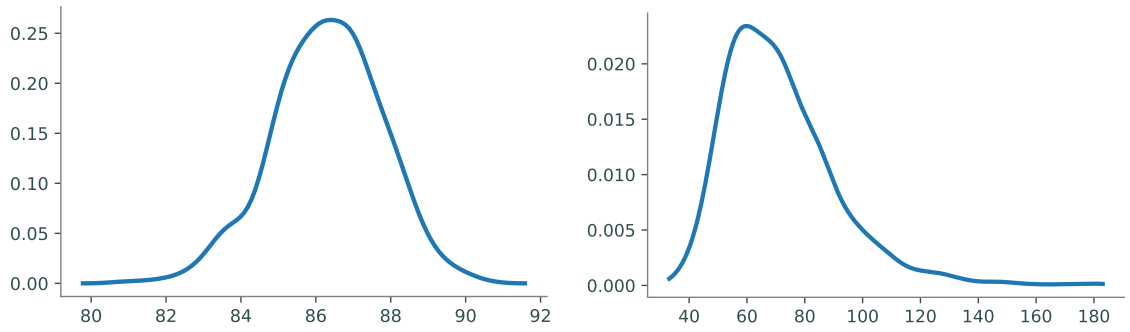
Problem 1. Write a function that accepts data $\mathbf{y}$, prior parameters ν, τ^2, α, and β, and an integer n. Use Gibbs sampling to generate n samples of μ and σ^2 for the exam scores problem.

Test your sampler with priors $\nu = 80$, $\tau^2 = 16$, $\alpha = 3$, and $\beta = 50$, collecting 1000 samples. Plot your samples of μ and your samples of σ^2. They should each to converge quickly.

We'd like to look at the posterior marginal distributions for μ and σ^2. To plot these from the samples, use a kernel density estimator from `scipy.stats`. If our samples of μ are called `mu_samples`, then we can do this with the following code.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> from scipy.stats import gaussian_kde

>>> mu_kernel = gaussian_kde(mu_samples)
>>> x = np.linspace(min(mu_samples) - 1, max(mu_samples) + 1, 200)
>>> plt.plot(x, mu_kernel(x))
>>> plt.show()
```



(a) Posterior distribution of μ.

(b) Posterior distribution of σ^2.

Figure 3.1: Posterior marginal probability densities for μ and σ^2.

Keep in mind that the plots above are of the posterior distributions of the *parameters*, not of the scores. If we would like to compute the posterior distribution of a new exam score $\tilde{y}$ given our data $\mathbf{y}$ and prior parameters, we compute what is known as the *posterior predictive distribution*:

$$\mathbb{P}(\tilde{y}|\mathbf{y}, \lambda) = \int_{\Theta} \mathbb{P}(\tilde{y}|\Theta)\mathbb{P}(\Theta|\mathbf{y}, \lambda)d\Theta$$

where Θ denotes our parameters (in our case μ and σ^2) and λ denotes our prior parameters (in our case ν, τ^2, α, and β).

Rather than actually computing this integral for each possible $\tilde{y}$, we can do this by sampling scores from our parameter samples. In other words, sample

$$\tilde{y}_{(t)} \sim N(\mu_{(t)}, \sigma^2_{(t)})$$

for each sample pair $\mu_{(t)}, \sigma^2_{(t)}$. Now we have essentially drawn samples from our posterior predictive distribution, and we can use a kernel density estimator to plot this distribution from the samples.

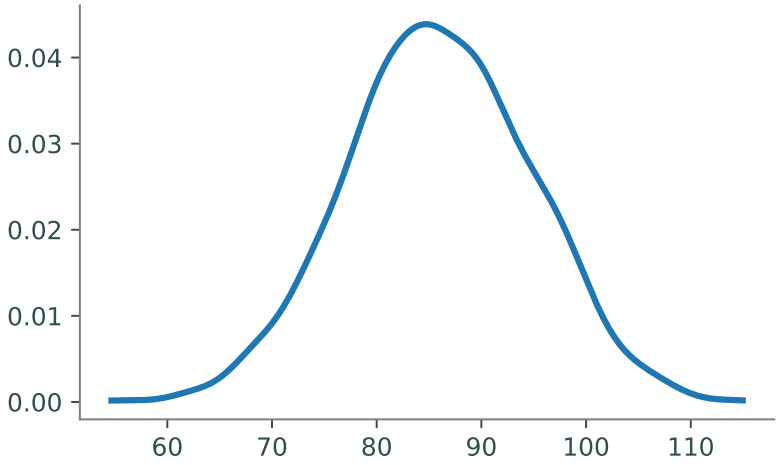


Figure 3.2: Predictive posterior distribution of exam scores.

Problem 2. Plot the kernel density estimators for the posterior distributions of μ and σ^2. You should get plots similar to those in Figure 3.1.

Next, use your samples of μ and σ^2 to draw samples from the posterior predictive distribution. Plot the kernel density estimator of your sampled scores. Compare your plot to Figure 3.2.

Latent Dirichlet Allocation

Gibbs sampling can be applied to an interesting problem in natural language processing (NLP): determining which topics are prevalent in a document. *Latent Dirichlet Allocation* (LDA) is a generative model for a collection of text documents. It supposes that there is some fixed vocabulary (composed of V distinct terms) and K different topics, each represented as a probability distribution ϕ_k over the vocabulary, each with a Dirichlet prior β. This means $\phi_{k,v}$ is the probability that topic k is represented by vocabulary term v.

With the vocabulary and topics chosen, the LDA model assumes that we have a set of M documents (each "document" may be a paragraph or other section of the text, rather than a "full" document). The m-th document consists of N_m words, and a probability distribution θ_m over the topics is drawn from a Dirichlet distribution with parameter α. Thus $\theta_{m,k}$ is the probability that document m is assigned the label k. If $\phi_{k,v}$ and $\theta_{m,k}$ are viewed as matrices, their rows sum to one.

We will now iterate through each document in the same manner. Assume we are working on document m, which you will recall contains N_m words. For word n, we first draw a topic assignment $z_{m,n}$ from the categorical distribution θ_m, and then we draw a word $w_{m,n}$ from the categorical distribution $\phi_{z_{m,n}}$. Throughout this implementation, we assume α and β are scalars. In summary, we have

1. Draw $\phi_k \sim \text{Dir}(\beta)$ for $1 \leq k \leq K$.

2. For $1 \leq m \leq M$:

   (a) Draw $\theta_m \sim \text{Dir}(\alpha)$.

   (b) Draw $z_{m,n} \sim \text{Cat}(\theta_m)$ for $1 \leq n \leq N_m$.

   (c) Draw $w_{m,n} \sim \text{Cat}(\phi_{z_{m,n}})$ for $1 \leq n \leq N_m$.

We end up with n words which represent document m. Note that these words are *not* necessarily distinct from one another; indeed, we are most interested in the words that have been repeated the most.

This is typically depicted with graphical plate notation as in Figure 3.3.

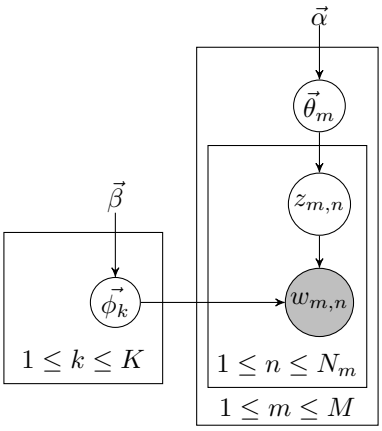


Figure 3.3: Graphical plate notation for LDA text generation.

In the plate model, only the variables $w_{m,n}$ are shaded, signifying that these are the only observations visible to us; the rest are latent variables. Our goal is to estimate each ϕ_k and each θ_m. This will allow us to understand what each topic is, as well as understand how each document is distributed over the K topics. In other words, we want to predict the topic of each document, and also which words best represent this topic. We can estimate these well if we know $z_{m,n}$ for each m, n, collectively referred to as $\mathbf{z}$. Thus, we need to sample $\mathbf{z}$ from the posterior distribution $\mathbb{P}(\mathbf{z}|\mathbf{w}, \alpha, \beta)$, where $\mathbf{w}$ is the collection words in the text corpus. Unsurprisingly, it is intractable to sample directly from the joint posterior distribution. However, letting $\mathbf{z}_{-(m,n)} = \mathbf{z} \setminus \{z_{m,n}\}$, the conditional posterior distributions

$$\mathbb{P}(z_{m,n} = k|\mathbf{z}_{-(m,n)}, \mathbf{w}, \alpha, \beta)$$

have nice, closed form solutions, making them easy to sample from.

These conditional distributions have the following form:

$$\mathbb{P}(z_{m,n} = k|\mathbf{z}_{-(m,n)}, \mathbf{w}, \alpha, \beta) \propto \frac{(n_{(k,m,\cdot)}^{-(m,n)} + \alpha)(n_{(k,\cdot,w_{m,n})}^{-(m,n)} + \beta)}{n_{(k,\cdot,\cdot)}^{-(m,n)} + V\beta}$$

where

$$
\begin{aligned}
n_{(k,m,\cdot)} &= \text{ the number of words in document } m \text{ assigned to topic } k \\
n_{(k,\cdot,v)} &= \text{ the number of times term } v = w_{m,n} \text{ is assigned to topic } k \\
n_{(k,\cdot,\cdot)} &= \text{ the number of times topic } k \text{ is assigned in the corpus} \\
n^{-(m,n)}_{(k,m,\cdot)} &= n_{(k,m,\cdot)} - \mathbf{1}_{z_{m,n}=k} \\
n^{-(m,n)}_{(k,\cdot,v)} &= n_{(k,\cdot,v)} - \mathbf{1}_{z_{m,n}=k} \\
n^{-(m,n)}_{(k,\cdot,\cdot)} &= n_{(k,\cdot,\cdot)} - \mathbf{1}_{z_{m,n}=k}
\end{aligned}
$$

Thus, if we simply keep track of these count matrices, then we can easily create a Gibbs sampler over the topic assignments. This is actually a particular class of samplers known as *collapsed Gibbs samplers*, because we have collapsed the sampler by integrating out θ and ϕ.

We have provided for you the structure of a Python object `LDACGS` with several methods, listed at the end of the lab. The object is already defined to have attributes `n_topics`, `documents`, `vocab`, `alpha`, and `beta`, where `vocab` is a list of strings (terms), and documents is a list of dictionaries (a dictionary for each document). Each entry in dictionary m is of the form $n : w$, where w is the index in `vocab` of the n^{th} word in document m.

Throughout this lab we will guide you through writing several more methods in order to implement the Gibbs sampler. The first step is to initialize our assignments, and create the count matrices $n_{(k,m,\cdot)}, n_{(k,\cdot,v)}$ and vector $n_{(k,\cdot,\cdot)}$.

Problem 3. Complete the method `initialize()`. By randomly assigning initial topics, fill in the count matrices and topic assignment dictionary. In this method, you will initialize the count matrices (among other things). Note that the notation provided in the code is slightly different than that used above. Be sure to understand how the formulae above connect with the code.

To be explicit, you will need to initialize nmz, nzw, and nz to be zero arrays of the correct size. Then, in the second for loop, you will assign z to be a random integer in the correct range of topics. In the increment step, you need to figure out the correct indices to increment by one for each of the three arrays. Finally, assign *topics* as given.

The next method we need to write fully outlines a sweep of the Gibbs sampler.

Problem 4. Complete the method `_sweep()`, which needs to iterate through each word of each document. It should call on the method `_conditional()` to get the conditional distribution at each iteration.

Note that the first part of this method will undo what `initialize()` did. Then we will use the conditional distribution (instead of the uniform distribution we used previously) to pick a more accurate topic assignment. Finally, the latter part repeats what we did in `initialize()`, but does so using this more accurate topic assignment.

We are now prepared to write the full Gibbs sampler.

Problem 5. Complete the method `sample()`. The argument *filename* is the name and location of a .txt file, where each line is considered a document. The corpus is built by method `buildCorpus`, and stopwords are removed (if argument *stopwords* is provided). Burn in the Gibbs sampler, computing and saving the log-likelihood with the method `_loglikelihood`. After the burn in, iterate further, accumulating your count matrices, by adding `nzw` and `nmz` to `total_nzw` and `total_nmz` respectively, where you only add every $sample\_rate^{th}$ iteration. Also save each log-likelihood.

You should now have a working Gibbs sampler to perform LDA inference on a corpus. Let's test it out on one of Ronald Reagan's State of the Union addresses, found in `reagan.txt`.

Problem 6. Create an `LDACGS` object with 20 topics, letting α and β be the default values. Run the Gibbs sampler, with a burn in of 100 iterations, accumulating 10 samples, only keeping the results of every 10th sweep. Use `stopwords.txt` as the stopwords file.

Plot the log-likelihoods. How long did it take to burn in?

We can estimate the values of each ϕ_k and each θ_m as follows:

$$\widehat{\theta}_{m,k} = \frac{n_{(k,m,\cdot)} + \alpha}{K \cdot \alpha + \sum_{k=1}^{K} n_{(k,m,\cdot)}}$$
$$\widehat{\phi}_{k,v} = \frac{n_{(k,\cdot,v)} + \beta}{V \cdot \beta + \sum_{v=1}^{V} n_{(k,\cdot,v)}}$$

We have provided methods `phi` and `theta` that do this for you. We often examine the topic-term distributions ϕ_k by looking at the n terms with the highest probability, where n is small (say 10 or 20). We have provided a method `topterms` which does this for you.

Problem 7. Using the methods described above, examine the topics for Reagan's addresses. As best as you can, come up with labels for each topic. If $ntopics = 20$ and $n = 10$, we will get the top 10 words that represent each of the 20 topics; for each topic, decide what these ten words jointly represent.

We can use $\widehat{\theta}$ to find the paragraphs in Reagan's addresses that focus the most on each topic. The documents with the highest values of $\widehat{\theta}_k$ are those most heavily focused on topic k. For example, if you chose the topic label for topic p to be *the Cold War*, you can find the five highest values in $\widehat{\theta}_p$, which will tell you which five paragraphs are most centered on the Cold War.

Let's take a moment to see what our Gibbs sampler has accomplished. By simply feeding in a group of documents, and with no human input, we have found the most common topics discussed, which are represented by the words most frequently used in relation to that particular topic. The only work that the user has done is to assign topic labels, saying what the words in each group have in common. As you may have noticed, however, these topics may or may not be *relevant* topics. You might have noticed that some of the most common topics were simply English particles (words such as *a*, *the*, *an*) and conjunctions (*and*, *so*, *but*). Industrial grade packages can effectively remove such topics so that they are not included in the results.

Additional Material

LDACGS Source Code

```
class LDACGS:
    """Do LDA with Gibbs Sampling."""

    def __init__(self, n_topics, alpha=0.1, beta=0.1):
        """Initialize system parameters."""
        self.n_topics = n_topics
        self.alpha = alpha
        self.beta = beta

    def buildCorpus(self, filename, stopwords_file=None):
        """Read the given filename and build the vocabulary."""
        with open(filename, 'r') as infile:
            doclines = [line.rstrip().lower().split(' ') for line in infile]
        n_docs = len(doclines)
        self.vocab = list({v for doc in doclines for v in doc})
        if stopwords_file:
            with open(stopwords_file, 'r') as stopfile:
                stops = stopfile.read().split()
            self.vocab = [x for x in self.vocab if x not in stops]
            self.vocab.sort()
        self.documents = []
        for i in range(n_docs):
            self.documents.append({})
            for j in range(len(doclines[i])):
                if doclines[i][j] in self.vocab:
                    self.documents[i][j] = self.vocab.index(doclines[i][j])

    def initialize(self):
        """Initialize the three count matrices."""
        self.n_words = len(self.vocab)
        self.n_docs = len(self.documents)

        # Initialize the three count matrices.
        # The (i,j) entry of self.nmz is the number of words in document i ↩
            assigned to topic j.
        self.nmz = np.zeros((self.n_docs, self.n_topics))
        # The (i,j) entry of self.nzw is the number of times term j is assigned↩
             to topic i.
        self.nzw = np.zeros((self.n_topics, self.n_words))
        # The (i)-th entry is the number of times topic i is assigned in the ↩
            corpus.
        self.nz = np.zeros(self.n_topics)

        # Initialize the topic assignment dictionary.
        self.topics = {} # key-value pairs of form (m,i):z
```

```
        for m in range(self.n_docs):
            for i in self.documents[m]:
                # Get random topic assignment, i.e. z = ...
                # Increment count matrices
                # Store topic assignment, i.e. self.topics[(m,i)]=z
                raise NotImplementedError("Problem 3 Incomplete")

    def sample(self,filename, burnin=100, sample_rate=10, n_samples=10, ↩
        stopwords=None):
        self.buildCorpus(filename, stopwords)
        self.initialize()
        self.total_nzw = np.zeros((self.n_topics, self.n_words))
        self.total_nmz = np.zeros((self.n_docs, self.n_topics))
        self.logprobs = np.zeros(burnin + sample_rate*n_samples)
        for i in range(burnin):
            # Sweep and store log likelihood.
            raise NotImplementedError("Problem 5 Incomplete")
        for i in range(n_samples*sample_rate):
            # Sweep and store log likelihood
            raise NotImplementedError("Problem 5 Incomplete")
            if not i % sample_rate:
                # accumulate counts
                raise NotImplementedError("Problem 5 Incomplete")

    def phi(self):
        phi = self.total_nzw + self.beta
        self._phi = phi / np.sum(phi, axis=1)[:,np.newaxis]

    def theta(self):
        theta = self.total_nmz + self.alpha
        self._theta = theta / np.sum(theta, axis=1)[:,np.newaxis]

    def topterms(self,n_terms=10):
        self.phi()
        self.theta()
        vec = np.atleast_2d(np.arange(0,self.n_words))
        topics = []
        for k in range(self.n_topics):
            probs = np.atleast_2d(self._phi[k,:])
            mat = np.append(probs,vec,0)
            sind = np.array([mat[:,i] for i in np.argsort(mat[0])]).T
            topics.append([self.vocab[int(sind[1,self.n_words - 1 - i])] for i ↩
                in range(n_terms)])
        return topics

    def toplines(self,n_lines=5):
        lines = np.zeros((self.n_topics,n_lines))
        for i in range(self.n_topics):
```

```
            args = np.argsort(self._theta[:,i]).tolist()
            args.reverse()
            lines[i,:] = np.array(args)[0:n_lines] + 1
        return lines

    def _removeStopwords(self, stopwords):
        return [x for x in self.vocab if x not in stopwords]

    def _conditional(self, m, w):
        dist = (self.nmz[m,:] + self.alpha) * (self.nzw[:,w] + self.beta) / (↩
            self.nz + self.beta*self.n_words)
        return dist / np.sum(dist)

    def _sweep(self):
        for m in range(self.n_docs):
            for i in self.documents[m]:
                # Retrieve vocab index for i-th word in document m.
                # Retrieve topic assignment for i-th word in document m.
                # Decrement count matrices.
                # Get conditional distribution.
                # Sample new topic assignment.
                # Increment count matrices.
                # Store new topic assignment.
                raise NotImplementedError("Problem 4 Incomplete")

    def _loglikelihood(self):
        lik = 0

        for z in range(self.n_topics):
            lik += np.sum(gammaln(self.nzw[z,:] + self.beta)) - gammaln(np.sum(↩
                self.nzw[z,:] + self.beta))
            lik -= self.n_words * gammaln(self.beta) - gammaln(self.n_words*↩
                self.beta)

        for m in range(self.n_docs):
            lik += np.sum(gammaln(self.nmz[m,:] + self.alpha)) - gammaln(np.sum↩
                (self.nmz[m,:] + self.alpha))
            lik -= self.n_topics * gammaln(self.alpha) - gammaln(self.n_topics*↩
                self.alpha)

        return lik
```