SDS386D: Monte Carlo Methods in Statistics **Cesar N. Yahia**
Homework 5 Prof. S.G. Walker

Take $n = 100$ samples from the density $g_0(x) = 3e^{-3x} \quad x > 0$
Toe estimate this density, use the model $g(x|w_M, M) = \sum_{j=1}^{M} w_{j,M} j e^{-jx}$ which is a mixture of exponential densities. The unknowns are M and the weights (w_M). The prior for w_M given M is Dirichlet with all parameters set to 1 s.t $f(w_{1M}, ...w_{MM}|M) \propto 1$. The prior for M is $f(M) \propto 1/(M-1)!$ for $M = 1, 2, ...$

(i)

write down the conditional $f(w^{(M)}|x_1, .., x_n, M)$.
without introducing labels d, the conditional $f(w^{(M)}|x_1, x_2.., x_n, M) \propto f(w^{(M)}|M) \prod_{i=1}^{n} \sum_{j=1}^{M} w_{j,M} j e^{-jx_i}$

This posterior is difficult to sample so we have to introduce a latent variable d_i which represents an association between the data point i and a component in the exponential mixture. When you introduce $d^{(M)}$, $w^{(M)}|M$ is isolated from the rest of the parameters in the model.

Therefore, $f(w^{(M)}|x_1, x_2.., x_n, d_1, d_2, .., d_n, M)$ reduces to $f(w^{(M)}|d_1, d_2, .., d_n, M)$
$f(w^{(M)}|d_1, d_2, .., d_n, M) \propto f(w^{(M)}|M) w_1^{n_1} w_2^{n_2}, .., w_M^{n_M} \propto w_1^{n_1} w_2^{n_2}, .., w_M^{n_M}$
where n_1 correspond to the number of data points i with label $d_i = 1$ referring to component 1. Similarly, $n_2, .., n_M$.

This implies that,
$f(w^{(M)}|M, d_1, d_2, .., d_n) \propto Dir(n_1 + 1, n_2 + 1, .., n_M + 1)$
Introducing the latent label variables d_i makes the Dirichlet a conjugate prior for $w^M|M$.

The numpy random library in Python has a built in function for sampling a Dirichlet given a list of parameters. This is used instead of explicitly using Gamma random variables to sample the Dirichlet.

(ii)

Now to write down $P(d_i = j|M, x_1, x_2, ..x_n, w_M)$ we recognize that this is proportional to the probability of the data point belonging to that component and the probability of that component.
In other words, $P(d_i = j|M, x_1, x_2, ..x_n, w_M) \propto w_{j,M} j e^{-jx_i}$
$P(d_i = j|M, x_i, w_M) = \frac{w_{j,M} j e^{-jx_i}}{\sum_{k=1}^{M} w_{k,M} k e^{-kx_i}}$
This can be written like that since we are conditioning on M and the corresponding weights are known.
To sample this, use the CDF method. Particularly, get the CDF for all the labels $d_i = \{1, 2, .., M\}$, then sample a uniform random variable, if the uniform is between $F(d = a)$ and $F(d = b)$ take $d = b$.

(iii)

To sample M using a Metropolis step, we have to introduce latent variables that make our model infinite dimensional. Specifically, as M changes, the dimension changes, and to ensure that we have the appropriate parameters as the dimension changes, we can introduce infinite latent variables. The model is now $g(x|w,M) = \sum_{j=1}^{M} w_{j,M} j e^{-jx} \prod_{k=1}^{M-1} f(w^{k-1}|w^{k}) \prod_{k=M}^{\infty} f(w^{k+1}|w^{k})$ where w contains w^M and all other latent variables w^{-M} corresponding to different number of components.

The resulting posterior is given by:
$f(w,M|data) \propto f(M) f(w^{(M)}|M) \prod_{i=1}^{n} \sum_{j=1}^{M} w_{j,M} j e^{-jx_i} \prod_{k=1}^{M-1} f(w^{(k-1)}|w^{(k)}) \prod_{k=M}^{\infty} f(w^{(k+1)}|w^{(k)})$

To sample M from the posterior, you can use a Metropolis step with a random walk proposal. The proposal can be

$$q(M'|M) = \begin{cases} M' = M+1 & \text{probability } 1/2 \\ M' = M-1 & \text{probability } 1/2 \end{cases}$$

for $M > 1$ and $q(2|1) = 1$.
Effectively, we have a random walk proposal that does not go below 1.

Then, following the M-H procedure, sample u from $U(0,1)$ and evaluate $\alpha(M', M)$.
$\star$ If $M' = M+1$ then
$\alpha(M', M) = \min\{1, \frac{f(M+1) f(w^{(M+1)}|M+1) \prod_{i=1}^{n} \sum_{j=1}^{M+1} w_{j,M+1} j e^{-jx_i} f(w^{(M)}|w^{(M+1)})}{f(M) f(w^{(M)}|M) \prod_{i=1}^{n} \sum_{j=1}^{M} w_{j,M} j e^{-jx_i} f(w^{(M+1)}|w^{(M)})}\}$
This last expression for α can be evaluated if we know $w^{(M+1)}$. Therefore, in anticipation to moving to $M+1$, sample $w^{(M+1)}$ from $f(w^{(M+1)}|w^{(M)})$. I will define $f(w^{(M+1)}|w^{(M)})$ and $f(w^{(M)}|w^{(M+1)})$ after outlining the M-H procedure.

$\star$ If $M' = M-1$ then
$\alpha(M', M) = \min\{1, \frac{f(M-1) f(w^{(M-1)}|M-1) \prod_{i=1}^{n} \sum_{j=1}^{M-1} w_{j,M-1} j e^{-jx_i} f(w^{(M)}|w^{(M-1)})}{f(M) f(w^{(M)}|M) \prod_{i=1}^{n} \sum_{j=1}^{M} w_{j,M} j e^{-jx_i} f(w^{(M-1)}|w^{(M)})}\}$.
This last expression for α can be evaluated if we know $w^{(M-1)}$. Therefore, in anticipation to moving to $M-1$, sample $w^{(M-1)}$ from $f(w^{(M-1)}|w^{(M)})$. I will define $f(w^{(M-1)}|w^{(M)})$ and $f(w^{(M)}|w^{(M-1)})$ after outlining the M-H procedure.

Then, if $u < \alpha(M', M)$ accept the move and change M to M'. Otherwise, keep M at the next iteration at its current value.
To implement the proposals for w, if the dimension increases, then determine the new vector w by choosing a component j at random, and splitting w_j into ww_j and $(1-w)w_j$ with w coming from $p(w)$ which is $U(0,1)$. Then we have that $f(w^{(M+1)}|w^{(M)}) = \frac{1}{M} p(w) = \frac{1}{M}$
If the dimension decreases, we can pick two components at random and combine their weights to create a new vector of w. This gives that $f(w^{(M-1)}|w^{(M)}) = \frac{2}{M(M-1)}$

Remaining is to define α for special cases when $M+1 = 2$ or $M-1 = 1$
$\star$(1) $M' = 2$, $M = 1$, then $q(M'|M) = 1$ and $q(M|M') = 1/2$
Then if dimension is increasing:
$\alpha(M', M) = \min\{1, \frac{f(M+1) f(w^{(M+1)}|M+1) \prod_{i=1}^{n} \sum_{j=1}^{M+1} w_{j,M+1} j e^{-jx_i} f(w^{(M)}|w^{(M+1)})}{f(M) f(w^{(M)}|M) \prod_{i=1}^{n} \sum_{j=1}^{M} w_{j,M} j e^{-jx_i} f(w^{(M+1)}|w^{(M)})}(1/2)\}$
and if the dimension is decreasing:
$\alpha(M', M) = \min\{1, \frac{f(M-1) f(w^{(M-1)}|M-1) \prod_{i=1}^{n} \sum_{j=1}^{M-1} w_{j,M-1} j e^{-jx_i} f(w^{(M)}|w^{(M-1)})}{f(M) f(w^{(M)}|M) \prod_{i=1}^{n} \sum_{j=1}^{M} w_{j,M} j e^{-jx_i} f(w^{(M-1)}|w^{(M)})}(1/2)\}$

$\star(2)$ $M'=1$, $M=2$, then $q(M'|M)=1/2$ and $q(M|M')=1$
Then if dimension is increasing:
$\alpha(M',M)=\min\{1,\frac{f(M+1)f(w^{(M+1)}|M+1)\prod_{i=1}^{n}\sum_{j=1}^{M+1}w_{j,M+1}je^{-jx_i}f(w^{(M)}|w^{(M+1)})}{f(M)f(w^{(M)}|M)\prod_{i=1}^{n}\sum_{j=1}^{M}w_{j,M}je^{-jx_i}f(w^{(M+1)}|w^{(M)})}(2)\}$
and if the dimension is decreasing:
$\alpha(M',M)=\min\{1,\frac{f(M-1)f(w^{(M-1)}|M-1)\prod_{i=1}^{n}\sum_{j=1}^{M-1}w_{j,M-1}je^{-jx_i}f(w^{(M)}|w^{(M-1)})}{f(M)f(w^{(M)}|M)\prod_{i=1}^{n}\sum_{j=1}^{M}w_{j,M}je^{-jx_i}f(w^{(M-1)}|w^{(M)})}(2)\}$

In the above discussion, we did not consider the latent variables $d^{(M)}$. The reason is that if we are going to sample M conditioned on $d^{(M)}$, then we have to create latent variables d^{-M} that correspond to labels for different values of M and identify methods to generate a vector of labels $d^{(M')}$ for M' from the current vector $d^{(M)}$. Similar to the approach outlined for w, we need to maintain a vector for d that changes with M. However, the latent variables d_i are only introduced to help us sample $w|M$. Therefore, we can use collapsed Gibbs sampling where we do not need to sample $M|d,w$ and instead we sample $M|w$.

The Gibbs framework is then:
- start with arbitrary values for M and $d_i^{(M)}$
- sample $w^{(M)}$ from $f(w^{(M)}|M,d_1,d_2,..,d_n)$ as shown in part (i)
- sample M as in part (iii)
- sample d from multivariate bernoulli $P(d_i=j|M,x_i,w_M)$ as in part (ii)

The transition density (ignoring condition on data) is then $f(d'|M',w')f(M'|w')f(w'|d,M)$
We can show that this leads to sampling from the stationary as follows:
$\int\int\int f(d'|M',w')f(M'|w')f(w|d,M)f(w,M,d)dddwdM$
$=f(d'|M',w')f(M'|w')\int\int\int f(w'|d,M)f(w,M,d)dddwdM$
$=f(d'|M',w')f(M'|w')\int\int\int\frac{f(w',d,M)}{f(d,M)}f(w|M,d)f(d,M)dddwdM$
$=f(d'|M',w')f(M'|w')\int\int\int f(w',d,M)f(w|M,d)dwdddM$
$=f(d'|M',w')f(M'|w')\int\int f(w',d,M)dM\int f(w|M,d)dwdd$
$=f(d'|M',w')f(M'|w')\int\int f(w',d,M)dddM$
$=f(d'|M',w')f(M'|w')f(w)$
$=f(d',M',w')$

(iv)

Now implement the algorithm above in Python, first start by sampling data points from the exponential distribution with $\lambda=3$. Then, use those data points to sample the mixture posterior using the procedure outlined above.

The code was implemented in Python as follows:

```
"""
This code implements a Markov Chain Monte Carlo
algorithm for the Gibbs sampler in Homework 5 -
SDS386D Monte Carlo methods in statistics  (q1)

@cnyahia
"""
```

```
import numpy as np
import numpy.random as nprand
import matplotlib.pyplot as plt
import scipy.stats as stats
import math
import random
from random import randrange
import copy as cp

# define sample exponential
def sample_expon(lam, n):
        """
        this function samples exponentials
        :param lam: lambda
        :param n: size
        :return: samples
        """
        scale = 1.0 / lam
        samples = list(nprand.exponential(scale=scale, size=n))
        return samples


# sample w from a dirichlet distribution given lables
def sample_weights(labels, M):
        """
        samples the weights from a Dirichlet distribution given the
        list of labels and the size M
        :param labels: list of labels
        :param M: number of components
        :return: sample from a Dirichlet distribution
        """
        # count the number of labels referring to each component
        label_numbers = [0]*M
        for key, label in enumerate(list(range(1, M+1))):
        label_numbers[key] = labels.count(label)

        for key, label_number in enumerate(label_numbers):
                label_numbers[key] = label_number + 1

        weights = list(nprand.dirichlet(label_numbers))
        return weights


# sample the labels given the weights and M
def samples_labels(weights, M, data_points):
        """
        This method samples the labels given a list
        of weights and the number of components
        :param weights: weights sampled from Dirichlet
```

```
        :param M: number of components
        :param data_points: the generated data points
        :return: label of data points
        """
        label_samples = [0]*len(data_points)
        for data_point_key, data_point in enumerate(data_points):
        # probability of a specific data point belonging to each component
        probability_di = [0]*len(weights)
        for weight_key, weight in enumerate(weights):
                numerator = weight * (weight_key+1) * np.exp(-(weight_key+1)*dat
                denominator = evaluate_mixture(weights, data_point)
                probability_di[weight_key] = float(numerator) / denominator
                # sample the data_point from the multivariate bernoulli distribu
                label_samples[data_point_key] =
                 sample_multivar_bernoulli(probability_di)

        return label_samples


# evaluate the mixture
        def evaluate_mixture(weights, data_point):
        """
        evaluates the mixture model for the given weights
        and the value of x (data_point)
        :param weights: weights
        :param data_point: value of x
        :return: mixture output
        """
        result = 0
        for key, weight in enumerate(weights):
                result += weight * (key+1) * np.exp(-(key+1)*data_point)

        return result


# sample a multivariate bernoulli random variable given the probabilities
def sample_multivar_bernoulli(probabilities):
        """
        sample a multivariate bernoulli for labels 1,2,3,...
        :param probabilities: probabilities of 1,2,3...
        :return: sample
        """
        cumulative = [0]*len(probabilities)
        accumulate = 0
        for key, probability in enumerate(probabilities):
                accumulate += probability
                cumulative[key] = accumulate

        u = nprand.uniform(low=0, high=1)
```

```
        sample = 0
        for key, cumu in enumerate(cumulative):
                if u <= cumu:
                        sample = key + 1
                        break

        return sample


# define prior for M
def M_prior(M):
        """
        returns prior for M
        :param M: value of M
        :return: prior probability
        """
        prior = 1.0 / math.factorial(M-1)
        return prior


# define sampling of increased weight vector from transition
def w_plus(w):
        """
        generates an augmented weight vector for M+1
        from the transition for w
        :param w: current vector w
        :return: augmented vector w
        """
        new_weights = cp.deepcopy(w)
        random_index = randrange(0, len(w))
        split = nprand.uniform(low=0, high=1)
        new_w1 = split * new_weights[random_index]
        new_w2 = (1-split) * new_weights[random_index]
        new_weights[random_index:random_index+1] = (new_w1, new_w2)
        return new_weights


# define sampling of decreased weight vector from the transition
def w_minus(w):
        """
        generates a weight vector with one less item by combining 2
        :param w: current vector w
        :return: decreased vector w
        """
        new_weights = cp.deepcopy(w)
        two_weights = random.sample(w, 2)
        combined_weight = sum(two_weights)
        index = new_weights.index(two_weights[0])
        new_weights.remove(two_weights[0])
```

```
        new_weights.remove(two_weights[1])
        new_weights.insert(index, combined_weight)
        return new_weights


# def probability of moving to wM+1
def prob_w_plus(M):
        ans = 1.0 / M
        return ans


# def probability of decreasing size to wM-1
def prob_w_minus(M):
        ans = (2.0) / (M * (M - 1))
        return ans


# define alpha increasing
def alpha_increasing(w, M, data_points):
"""
returns alpha increasing
:param w: weights
:param M: M
:return: alpha
"""
new_weights = w_plus(w)
product_numerator = 1
for data_point in data_points:
product_numerator = product_numerator * evaluate_mixture(new_weights, data_point

numerator = M_prior(M+1) * product_numerator * prob_w_minus(M+1)
product_denominator = 1
for data_point in data_points:
product_denominator = product_denominator * evaluate_mixture(w, data_point)

denominator = M_prior(M) * product_denominator * prob_w_plus(M)
ratio = numerator / denominator
alpha = min(1.0, ratio)
return alpha


# define alpha decreasing
def alpha_decreasing(w, M, data_points):
"""
returns alpha decreasing
:param w: weights
:param M: M
:param data_points: data points
:return: alpha value
```

```
"""
new_weights = w_minus(w)
product_numerator = 1
for data_point in data_points:
product_numerator = product_numerator * evaluate_mixture(new_weights, data_point

numerator = M_prior(M-1) * product_numerator * prob_w_plus(M-1)

product_denominator = 1
for data_point in data_points:
product_denominator = product_denominator * evaluate_mixture(w, data_point)

denominator = M_prior(M) * product_denominator * prob_w_minus(M)
ratio = numerator / denominator
alpha = min(1.0, ratio)
return alpha


# define alpha increasing if current M is 1
def alpha_increasingM1(w, M, data_points):
"""
returns alpha increasing for special case
:param w: weights
:param M: M
:return: alpha
"""
new_weights = w_plus(w)
product_numerator = 1
for data_point in data_points:
product_numerator = product_numerator * evaluate_mixture(new_weights, data_point

numerator = M_prior(M + 1) * product_numerator * prob_w_minus(M + 1)
product_denominator = 1
for data_point in data_points:
product_denominator = product_denominator * evaluate_mixture(w, data_point)

denominator = M_prior(M) * product_denominator * prob_w_plus(M)
ratio = (numerator / denominator) * (0.5)
alpha = min(1.0, ratio)
return alpha


# define alpha decreasing for special case M1
def alpha_decreasingM1(w, M, data_points):
"""
returns alpha decreasing
:param w: weights
:param M: M
:param data_points: data points
```

```
:return: alpha value
"""
new_weights = w_minus(w)
product_numerator = 1
for data_point in data_points:
product_numerator = product_numerator * evaluate_mixture(new_weights, data_point

numerator = M_prior(M - 1) * product_numerator * prob_w_plus(M - 1)

product_denominator = 1
for data_point in data_points:
product_denominator = product_denominator * evaluate_mixture(w, data_point)

denominator = M_prior(M) * product_denominator * prob_w_minus(M)
ratio = (numerator / denominator) * (0.5)
alpha = min(1.0, ratio)
return alpha


# define alpha increasing if current M is 2
def alpha_increasingM2(w, M, data_points):
"""
returns alpha increasing for special case
:param w: weights
:param M: M
:return: alpha
"""
new_weights = w_plus(w)
product_numerator = 1
for data_point in data_points:
product_numerator = product_numerator * evaluate_mixture(new_weights, data_point

numerator = M_prior(M + 1) * product_numerator * prob_w_minus(M + 1)
product_denominator = 1
for data_point in data_points:
product_denominator = product_denominator * evaluate_mixture(w, data_point)

denominator = M_prior(M) * product_denominator * prob_w_plus(M)
ratio = (numerator / denominator) * 2.0
alpha = min(1.0, ratio)
return alpha


# define alpha decreasing for special case M1
def alpha_decreasingM2(w, M, data_points):
"""
returns alpha decreasing
:param w: weights
:param M: M
```

```
:param data_points: data points
:return: alpha value
"""
new_weights = w_minus(w)
product_numerator = 1
for data_point in data_points:
product_numerator = product_numerator * evaluate_mixture(new_weights, data_point

numerator = M_prior(M - 1) * product_numerator * prob_w_plus(M - 1)

product_denominator = 1
for data_point in data_points:
product_denominator = product_denominator * evaluate_mixture(w, data_point)

denominator = M_prior(M) * product_denominator * prob_w_minus(M)
ratio = (numerator / denominator) * 2.0
alpha = min(1.0, ratio)
return alpha


# implement the sampling from q
def sample_MH_prop(M):
"""
sample from q given the previous value
q is a random walk
:param xm: previous value
:return: sample
"""
die = nprand.uniform(low=0, high=1)
if M == 1:
sample = 2

else:
if die <= 0.5:
sample = M + 1
else:
sample = M - 1

return sample


# sample from mixture
def sample_mix_model(weights):
"""
samples the mixture
:param weights: weight parameters
:return:
"""
key = sample_multivar_bernoulli(weights)
```

```
value = sample_expon(key, 1)
return value


# evaluate exponential
def eval_expon(lam, x):
val = lam * np.exp(-lam * x)
return val


if __name__ == '__main__':
data_points = sample_expon(3, 100)
M = 3  # initial value for M
possible_labels = [1,2,3]
labels = [0]*len(data_points)
# set initial labels randomly
for key, data_point in enumerate(data_points):
labels[key] = random.choice(possible_labels)

samples_w = []
samples_M = []
samples_x = []
numiter = 1
alpha = 1
while numiter <= 500:
# sample w
weights = sample_weights(labels=labels, M=M)
samples_w.append(weights)

# sample M
M_star = sample_MH_prop(M)
if (M_star != 1) and (M_star != 2):
if M_star > M:
alpha = alpha_increasing(w=weights, M=M, data_points=data_points)
elif M_star < M:
alpha = alpha_decreasing(weights, M, data_points)
elif M_star == 1:
if M_star > M:
alpha = alpha_increasingM1(weights, M, data_points)
elif M_star < M:
alpha = alpha_decreasing(weights, M, data_points)
elif M_star == 2:
if M_star > M:
alpha = alpha_increasingM2(weights, M, data_points)
elif M_star < M:
alpha = alpha_decreasingM2(weights, M, data_points)
u = nprand.uniform(low=0, high=1)
if u < alpha:
M = M_star
```

```
# sample d
labels = samples_labels(weights, M, data_points)

# sample x
samples_x.append(sample_mix_model(weights))

numiter += 1

density_evaluate = list(np.arange(-5, 5.1, 0.05))
exponvals = [eval_expon(3, x) for x in density_evaluate]

plt.hist(samples_x, bins=50, facecolor='g', alpha=0.7, normed=True, label='predi
plt.plot(density_evaluate, exponvals, label='exponential')
plt.title("predictive density (sampled) vs. exponential")
plt.xlabel("x")
plt.ylabel("density")
plt.legend(loc='upper right')
plt.tight_layout()
plt.savefig('pred_.png')
plt.show()
```

(v)

I had a bug in the indexing and was not able to get results for this in time.