Statistical Methods in Applied Computer Science, DD2447

Fall 2019

Assignment 2

Josip Domazet & Ivan Landeka

January 15th 2020

Contents

3 The project

3.1 Gibbs sampler for the magic word

Question 1

We have generated data with following parameters:
alphabet = {"a", "b", "c", "d" }, $N = 10$, $M = 15$, $W = 4$,
$\boldsymbol{\alpha}^{\mathrm{T}} = [\,0.8 \quad 0.8 \quad 0.8 \quad 0.8\,]$ and $\boldsymbol{\alpha'}^{\mathrm{T}} = [\,1 \quad 1 \quad 1 \quad 1\,]$.
10 chains were used, each of which had 100 samples.
Starting positions for magic words were:

$$[\,5 \quad 9 \quad 3 \quad 10 \quad 8 \quad 10 \quad 12 \quad 12 \quad 4 \quad 9\,]$$

. Predicted starting position were:

$$[\,5 \quad 9 \quad 3 \quad 11 \quad 7 \quad 10 \quad 12 \quad 12 \quad 4 \quad 9\,]$$

. Since $\frac{8}{10}$ starting positions are correct, the accuracy is 80%.

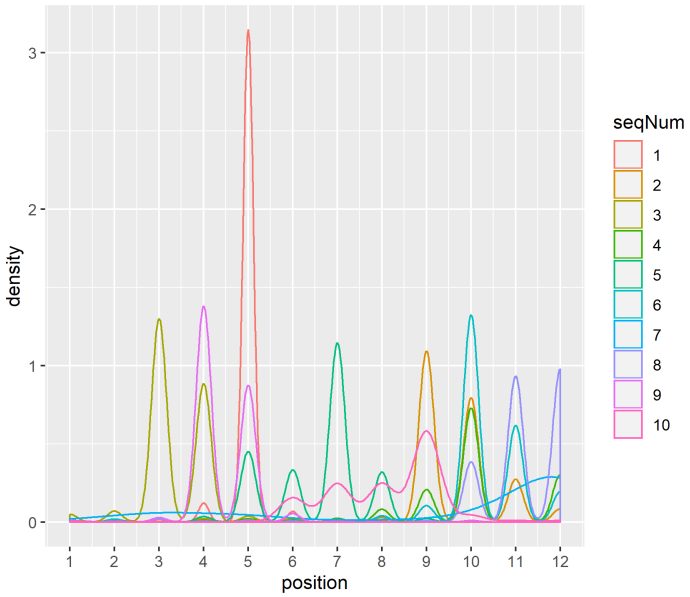


Figure 1: Posterior densities for the starting positions

From below Figure 2 we can see that potential scale reduction factor ($PSRF$) converges towards 1. That means that all chains have converged to the same target posterior distribution.

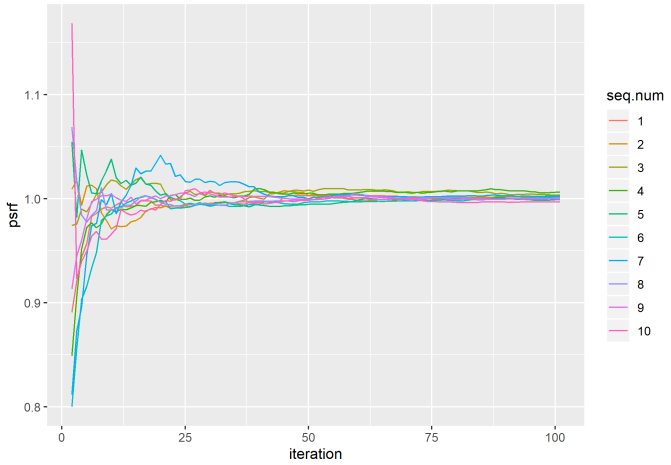


Figure 2: PSRF graph for synthetic data

By looking at Figure 2, we can conclude that convergence has been reached in about 50 iterations ($PSRF \longrightarrow 1.00$).

Question 2

10 chains were used, each with 100 samples. Each colored line on the figure represents one sequence. For this figure, $N = 20$ was used.

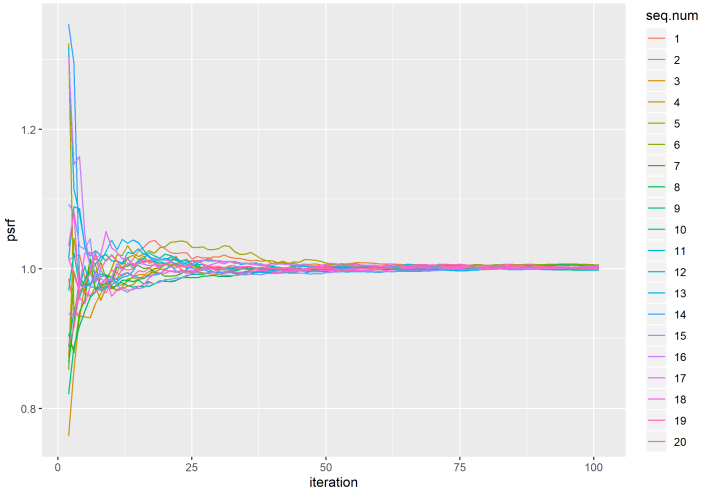


Figure 3: PSRF graph for *GitHub* data with $N = 20$

Note that Figure 3 is not in the same scale as Figure 2, i.e. the variance is higher in the former one (compare $y - axis$ values). However, we can still

observe the convergence of chains after 40 iterations.

The plot below was obtained using $N = 25$. Number of chains remained 10, but the number of samples per each chain was decreased to 20 due to increased computational cost.

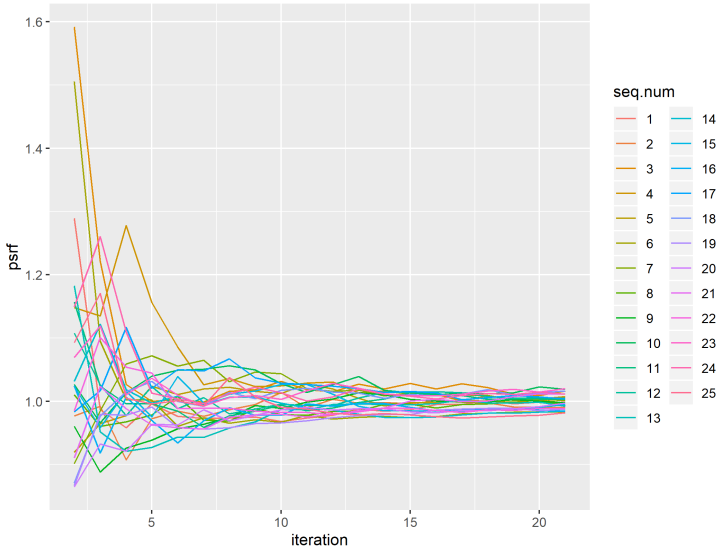


Figure 4: PSRF graph for *GitHub* data with $N = 25$

Convergence is still visible from Figure 4, even when only 20 samples are used per chains.

3.2 SMC for the stochastic volatility model

Question 3

The code for generating parameters and data is in the appendix.

Question 4

(i) The choice of proposal is $q(x_t \mid y_{1:t}) = \sum_i w_{t-1}^i p(x_t \mid x_{t-1}^i)$ since it enables us to compute weights using *already known* distributions which define the model: $w_t = p(y_t \mid x_t)$
$p(x_{1:t} \mid y_{1:t})$ is targeted with N particles $\{x_{1:t}^i\}_{i=1}^N$. Particles are obtained using proposal distribution $q(x_{1:t} \mid y_{1:t})$, so each particles gets it's importance weight

$$w_t^i = \frac{p(x_{1:t}^i \mid y_{1:t})}{q(x_{1:t}^i \mid y_{1:t})}$$

However, usually we know the target distribution up to normalizing constant. Normalized weights are obtained from non-normalized weights by equation: $w_t^i = \frac{\tilde{w}_t^i}{\sum_i \tilde{w}_t^i}$

IN order to use sequential relations, let proposal distribution be factorised as follows:

$$\begin{aligned} q_t(x_{1:t} \mid y_{1:t}) &= q_t(x_t \mid x_{t-1}, y_t) q_{t-1}(x_{1:t-1} \mid y_{1:t-1}) \\ &= q_1(x_1 \mid y_1) \prod_k q_k(x_k \mid x_{k-1}, y_k) \end{aligned}$$

This also factorises the weights

$$\begin{aligned} w_t^i = \frac{p(x_{1:t}^i \mid y_{1:t})}{q(x_{1:t}^i \mid y_{1:t})} &\propto \frac{p(y_t \mid x_t^i) p(x_t^i \mid x_{t-1}^i)}{q_t(x_t^i \mid x_{t-1}^i, y_t)} \frac{p(x_{1:t-1}^i \mid y_{1:t-1})}{q_{t-1}(x_{1:t-1}^i \mid y_{1:t-1})} \\ &= \frac{p(y_t \mid x_t^i) p(x_t^i \mid x_{t-1}^i)}{q_t(x_t^i \mid x_{t-1}^i, y_t)} w_{t-1}^i \end{aligned}$$

Questions 4, 5 and 6

(ii) Data: $x_{100} = -1.503388$

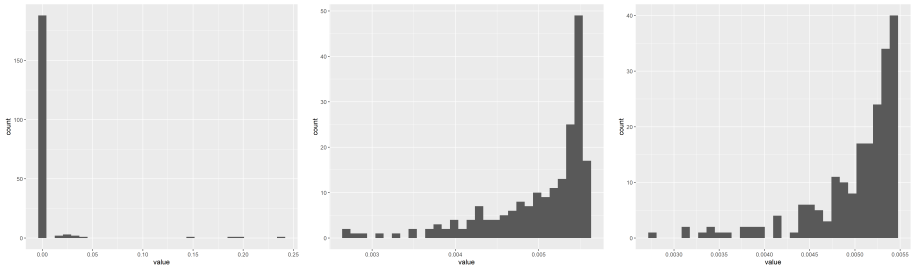
Method	$\hat{x}_{100}$	MSE
SIS	−1.472395	0.09525671
BPF multinomial	−1.410593	0.03265868
BPF stratified	−1.340721	0.03209481

Table 1: Comparison of three methods

(iii) Density

Figure 5: Weight histograms for three schemes

(a) SIS (b) BPF multinomial (c) BPF stratified

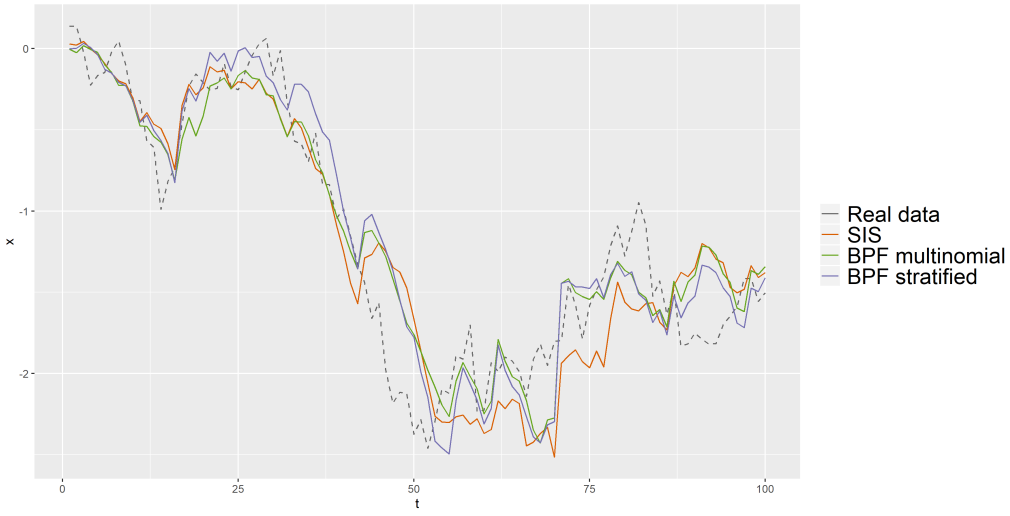


Scheme	$Var(\hat{w}_{100})$
SIS	8.08×10^{-5}
BPF multinomial	7.06×10^{-9}
BPF stratified	5.79×10^{-9}

Table 2: The empirical variance of normalized weights for three schemas

We can see from Figure 5 and Table 2 that SIS has the problem of weight degeneracy. Added multinomial and stratified resampling scheme have fixed that.

Figure 6: Visual comparison of three schemes



Question 7

We can see from Figure 6 that SIS perform slightly worse than bootstrap filters, especially around 50 and 70. Bootstrap filters tend to be more accurate since they solve weight decay problem with resampling. That also shows Table 1 with MSE column. Stratified bootstrap filter is the best overall since it scored the lowest $MSE(\hat{x}_{100})$. However, it's still very close to multinomial bootstrap filter.

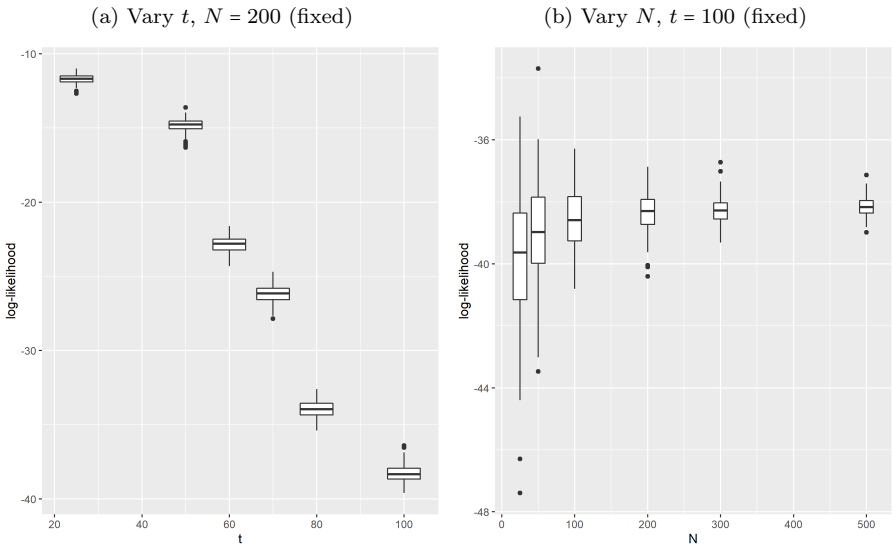
3.3 Stochastic volatility unknown parameters part I

Question 8

625 pairs of (β, σ) between 0 and 2 were used. After 10 runs of SMC for each pair, mean value of log-likelihood was stored and compared. The best parameter combination was $(\sigma = 0.161, \beta = 0.721)$ with $\overline{\log \hat{p}(y_{1:T})} = -183$, which is very close to real values of $\sigma = 0.16$ and $\beta = 0.64$.

Question 9

Figure 7: Varying N and T



From figure 7 we can conclude that the variance of the log-likelihood *increases* as t is increased. That makes sense because even in our daily life, it's easier to predict something that will happen in 3 days than in 30 days.
Furthermore, from figure 7 we can conclude that the variance of the log-likelihood *decreases* as N is increased. That also makes sense because the increase of the number of particles is analogous to increase of the samples. More samples make our estimate more confident.

Question 10

We used 10000 iterations and discarded first 5000 iterations as burn-in phase.

(i) Proposal distributions and MH acceptance ratio

(a) $\sigma^2_{new} = max\left\{0.0001, \mathcal{N}\left(\sigma^2_{old}, 0.01\right)\right\}$

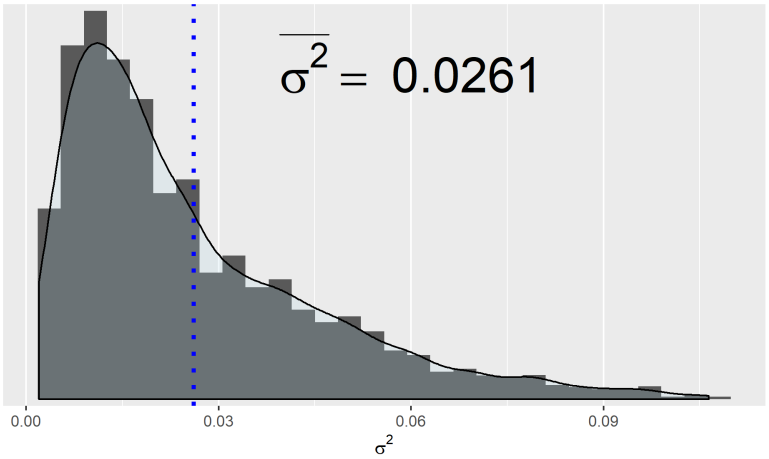
(b) $\beta^2_{new} = max\left\{0.01, \mathcal{N}\left(\beta^2_{old}, 0.2\right)\right\}$

Parameters σ and β need to be positive values, so I've used max function to avoid wasting iterations on getting positive values by assigning values a bit above zero. It's not equal to zero to avoid having all particles identical due to zero variance/standard deviation.

(c) Metropolis-Hastings acceptance ratio = 0.4486

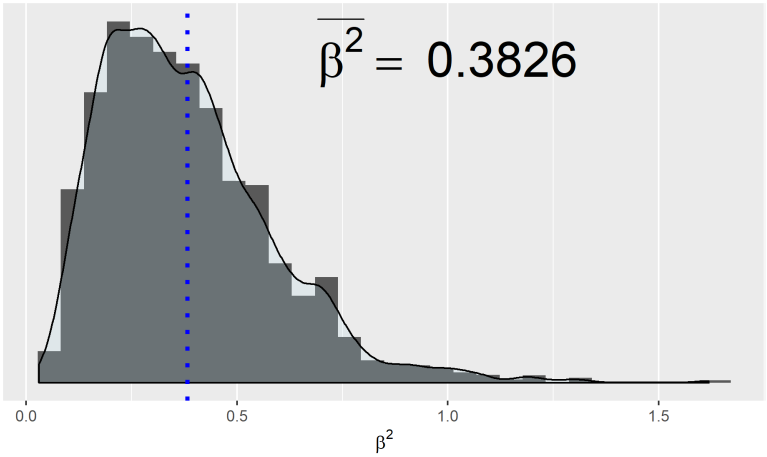
(ii) The evidence of success

Figure 8: The posterior distribution for σ^2



We can see from Figure 8 the posterior mean gives value pretty close to the real value of 0.16: $\hat{\sigma} = \sqrt{0.0261} = 0.1616$.

Figure 9: The posterior distribution for β^2



Analogous for σ^2, the posterior mean (Figure 9) gives $\hat{\beta} = \sqrt{0.3826} = 0.6185$. As a comparison, true value for $\beta = 0.64$. The obtained result for β is not as close as $\hat{\sigma}$ was to σ, but it's still very good.
These results are pretty credible since they're constructed with $N = 5000$ samples (after discarding first 5000 samples from total of 10000 samples).

3.4 Stochastic volatility unknown parameters part II

Initial parameter values were $\sigma^2_{initial} = 0.1$ and $\beta^2_{initial} = 0.1$.
Bootstrap filter with mutlinomial resampling was used with initial values of σ^2 and β^2 to obtain initial x values (hidden values) for the algorithm.

100 particles were used at the each iteration.

Figure 10: Particle Gibbs - The marginal distribution for σ^2

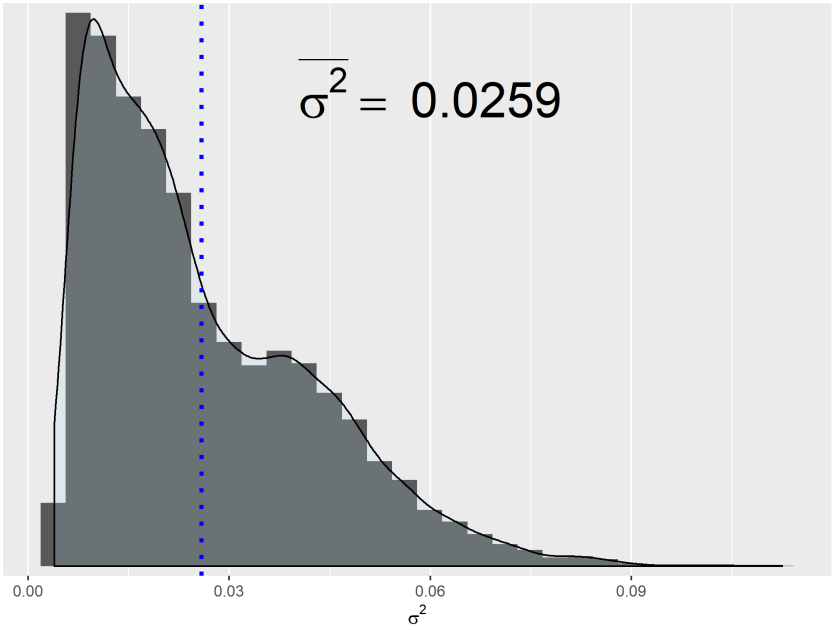


Figure 10 looks pretty close to Figure 8 obtained by Metropolis-Hastings algorithm. Figure 10 looks more smooth since it was constructed with 10000 more samples. The mean values are also very similar:
$\sigma^2_{PG} = 0.0261$ and $\sigma^2_{MH} = 0.0259$.
The discarding of the burn-in phase was important since not discarding it produces much worse results: $\sigma^2_{PG+burn-in} = 0.0421$.

Figure 11: The marginal distribution for β^2

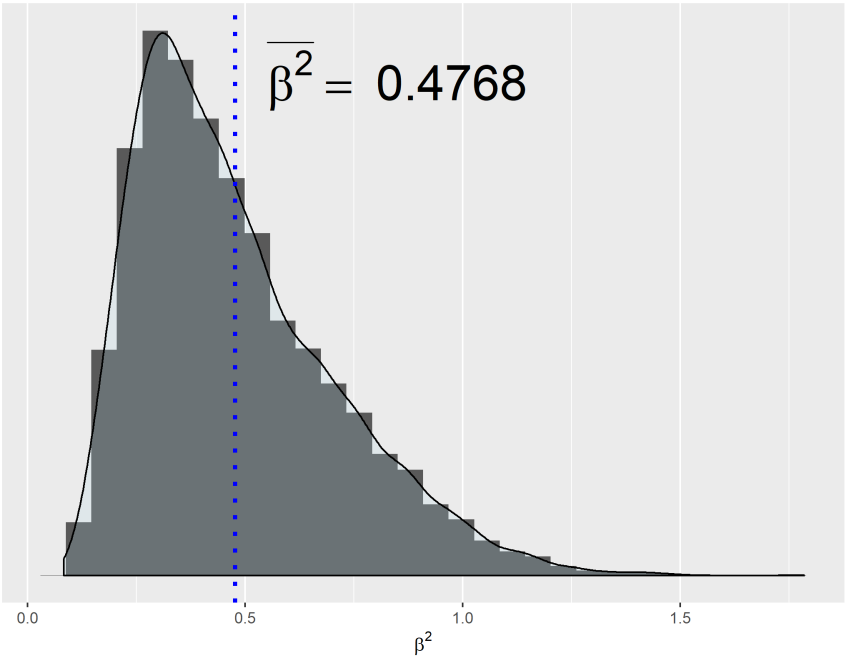


Figure 11 looks (by shape) close to Figure 9 obtained by Metropolis-Hastings algorithm. Figure 11 also looks more smooth since it was constructed with 10000 more samples. However, the mean values are not so close for β^2. Particle Gibbs approximation overestimated β^2:
$\beta^2_{PG} = 0.4768$ and $\beta^2_{MH} = 0.3826$

var	true value	PG var^2	MH var^2	PG *var*	MH *var*
σ	0.16	0.0258829	0.02609366	0.16088	0.161535
β	0.64	0.4767738	0.3825723	0.6904881	0.6185243

Table 3: Comparison of PG and MH algorithm

We can conclude from Table 3 that Particle Gibbs performed better in estimation of σ (0.16088 is closer to 0.16 than 0.161535), while Metropolis-Hastings algorithm perform better in estimation of β.
It should also be noted that the particle Gibbs algorithm was used with 20000 iterations total, after which first 5000 iterations were discarded as burn-in phase, i.e. to correct the bias for the initial samples. The Metropolis-Hastings algorithm was used with 10000 iterations total, also with first 5000 iterations being discarded as burn-in phase.

3.1 PyClone Light

Question 12

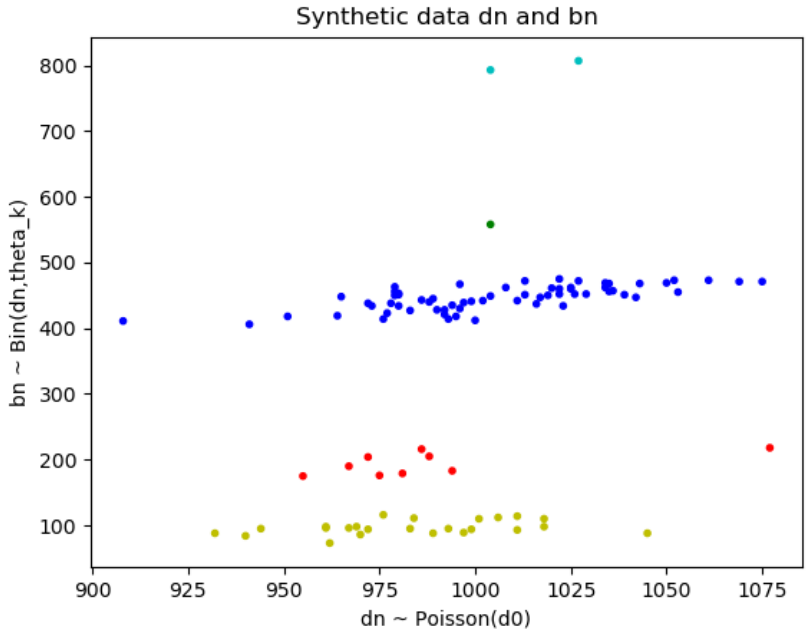
Implement forward simulator to generate synthetic data that accepts hyper parameters and the number of data points as inputs.

To generate the data, we first used the $GEM(\alpha)$ distribution to generate the mixture weights for classes (i.e. π). The GEM distribution in our case can be described as a sequence of random variables

$$\pi_1 = U_1, \pi_n = (1 - U_2)(1 - U_2)...(1 - Un - 1)U_n, \tag{1}$$

where the U_i is generated from $Beta(1, \alpha)$ distribution. According to those mixture weights the component labels (Z_n) for the specified number of data points are drawn (each cluster has a probability of its mixture weight). To generate the data from Binomial distribution we first need to get the parameters d^n (number of trials) and ϕ^{Z_n} (probability of success). We draw d_n from $Poisson(d_0)$ distribution (for every sample), where d_0 is an input argument to the function. The probability of success is drawn from $Beta(a0, a1)$ distribution for every mixture weight π_i. At last we generate b_n from $Binomial(d^n, \phi^{Z_n})$.

Figure 12: Dataset



The easiest way to visualize the mixtures and their corresponding samples inside the dataset component labels is a scatter plot of the data. This is an example of a 100 samples generated from 5 different mixtures with hyperparameters α = 1,a_0 = a_1 = 1, and d_0 = 1000 (Figure 12). We can see that a majority of samples belongs to the blue group, precisely 64 of them. That is not surprising because the mixture weight of this particular mixture is 0.656. As that mixture weight took a large portion of the probability, the other ones are not so often.

The generation of data is implemented inside the method *generate_data*() and *gem*().

Question 13

Implement collapsed Gibbs sampler for this DPMM. Test it on synthetic data with the following values of the hyper parameters: α = 1 , a0 = a1 = 1, and d0 = 1000. Provide a figure depicting accuracy of clustering.

The initialization of the clusters is implemented as a chinese restaurant process, where N_{th} customer randomly chooses his seat on the new table or on the non-empty table with the next probability :

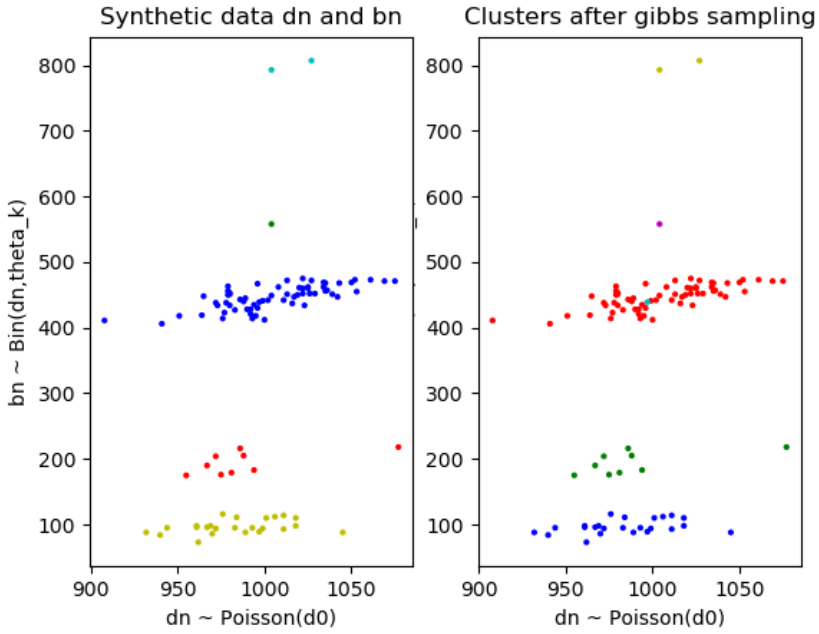
$$\mathrm{p}(\mathrm{z}_n = k|z_{-n}) = \begin{cases} \dfrac{N_k}{N+\alpha-1}, & \text{if } N_k \geq 1 \\ \dfrac{\alpha}{\alpha+N-1}, & \text{otherwise.} \end{cases}$$

After initialization of the clusters, we perform Collapsed Gibbs Sampling for this DPMM. For each sample, we compute the posterior predictive probability for each of the known clusters and the prior predictive for it to belong to a new cluster. Beta and Binomial distributions are conjugate, so the posterior predictive of a sample belonging to a known cluster is calculated as :

$$p(z_n = k|D) = \binom{d_n}{b_n} \frac{B(b_n + a + r, b + d_n - b_n + n - r)}{B(a + r, b + n)}, \tag{2}$$

where r is a sum of all the b_n that belong to cluster k, and n is a sum of all the d_n that belong to a cluster k. To calculate the probability of assigning the datapoint to a new cluster is easier as we don't have to use the n and r mentioned above.It is important to mention that we need to use the log of all those probabilities in order to evade overflow. Finally, after going over all of the samples, one iteration of Collapsed Gibbs sampler is implemented

Figure 13: Dataset



The accuracy of clustering for the dataset that we generated above is shown in Figure 13 . We clearly see that the majority of data points that were generated by the same mixture are in the same cluster computed by the model. Measure of accuracy that can be used in clustering problems is Rand index explained in question 14, it's value in this example is 0.9873.

Question 14

The number of clusters can be controlled via the parameter α. Study how the performance deteriorates with increasing number of clusters. How is the number of data points affecting the performance?

Because we might not get the same component label number for a certain cluster, and the number of clusters that the model computes can also be different, we cannot actually use any standard measures of accuracy. Clustering performance evalutation often uses a measure called Rand index which is calculated by the next formula :

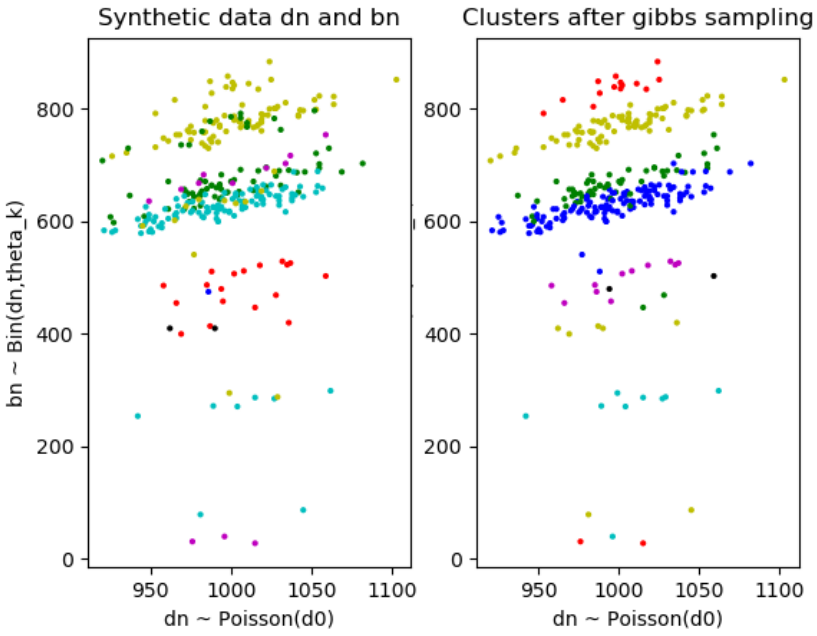
$$R = \frac{a+b}{\binom{N}{2}} \in [0,1], \tag{3}$$

where $\boldsymbol{a}$ is the number of equally labeled sample pairs that were also generated by the **same** mixture, and $\boldsymbol{b}$ is the number of differently clustered pairs that

were generated by **different** mixtures in the forward generation. N signifies the number of samples in total. This measure is exactly one if every sample is in a cluster only with samples that were generated from the same mixture.

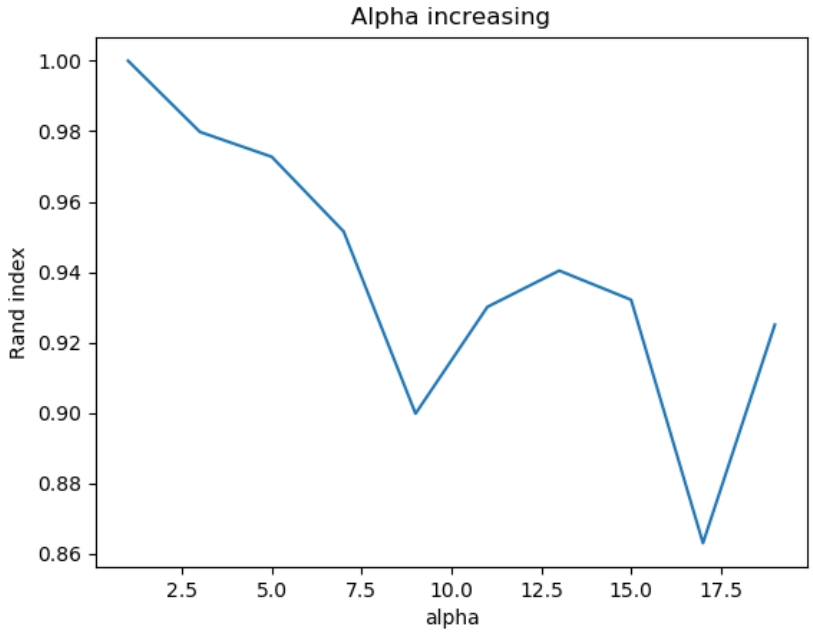
Increasing the parameter α means that we will draw the random variable U_i from a beta distribution that is skewed to the right, which tends to generate values that are more close to zero. Therefore, proportions of probability that will be assigned to each successive mixture weight in order will be smaller than those generated from a less skewed distribution. Ultimately, this results in more mixtures that will have significant weights (probability).
Intuitively, increasing number of clusters could decrease the performance of the algorithm. In this case, clusters are determined by probability parameters for the Binomial distribution ϕ generated from an independent Beta distribution. Since there is a bigger number of clusters, it is more likely that some of the parameters ϕ are going to be similar to each other so they will be harder to distinguish for the model. This type of behaviour is shown in Figure 14 where we have 300 samples generated with hyperparameter $\alpha = 5$ and 100 iterations. There are more clusters than in Figure 13, and some of them are mixed with each other. The Rand index performance measure is 0.859. If we only look at the Figure 14 without knowing anything about the data, we would say that the clustering algorithm works fine.

Figure 14: Increased $\alpha = 5$



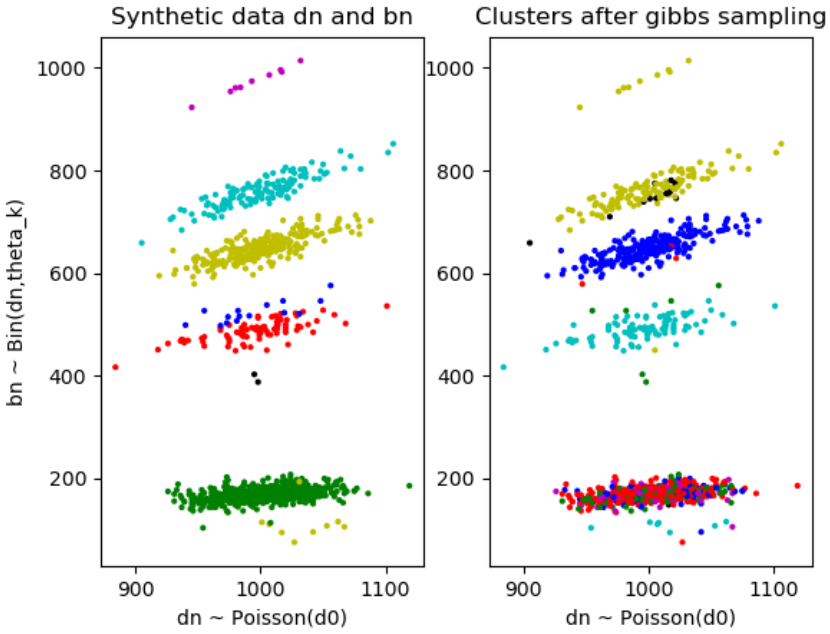
Plot of Rand index performance measure given increasing alpha for *iterations* = 100, N = 100 (Figure 15) :

Figure 15: Increasing of α



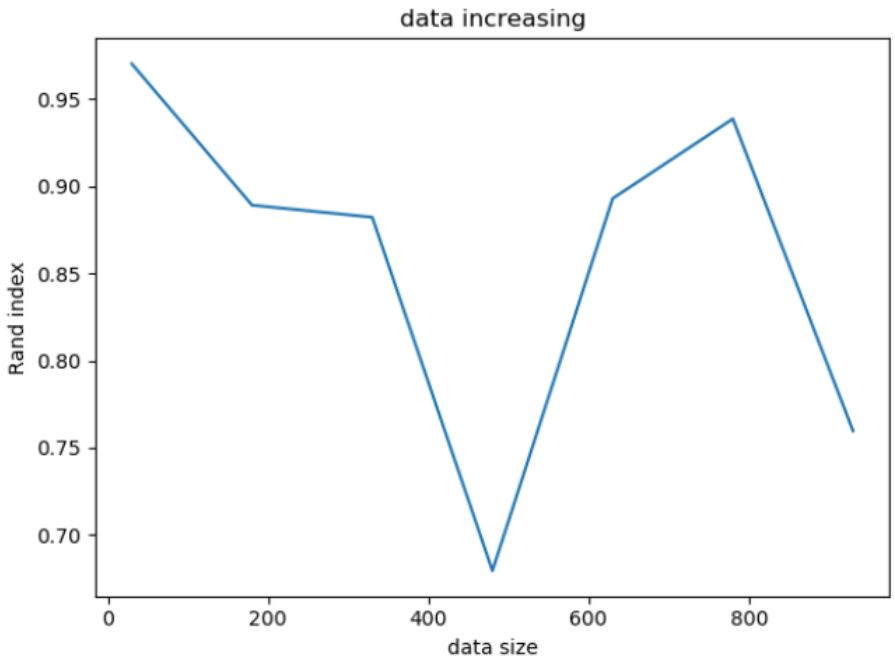
The results of clustering for α = 2 ,N = 2000, *iterations* = 30 is visualized in Figure 16. If we have a lot of data, if the clusters have lower variance and arent mixing together, we should get better results given that a lot of data would make the cumulative n and r really big that we use in posterior predictive really big. Also, if we have a lot of data, it is really important that we have a sufficient number of iterations so that Collapsed Gibbs Sampler converges. Otherwise, we might get slightly worse results, as shown in Figure 16.

Figure 16: Increased N



The performance regarding the data size shown in Figure 17. In the implementation, we keep the hyperparameters of DPMM fixed. The data is independently generated, but we can still see some improvements on the performance. Since the parapmeter ϕ can be generated differently every time, the performance is depending on it.

Figure 17: Performance regarding α and size of data



3.2 Predicting a relation based on Indian Buffet Process

Question 15

Implement an MCMC algorithm for this model.

We start by initializing **Z** from an IBP (we assume the hyperparameter is given) and randomly initializing matrix **W** given the number of features. We will use Metropolis-Hastings algorithm to propose a row of **Z** given **W**, and in the next step sample **W** by sequentially proposing each weight of the weight matrix. The **Z** proposal samples a single row of a matrix, considering that the customer that we are resampling is the last one to enter the restaurant. As we know, the probability of activating a certain feature that is present in other objects

$$p(z_{ik} = 1|Z_{-i}) = \frac{m_k}{N} \tag{4}$$

,where N is the number of objects, and m_k the number of objects that also have that feature. Additionally, after resampling the existing columns for a certain row, we propose $p \sim Poisson(\frac{\alpha}{N})$ new columns as it was the case when generating the data in IBP, and sample weights for that particular feature from $\mathcal{N}(0, \sigma^2)$.
Given the full set of observations **Y**,we wish to infer the posterior distribution of the feature matrix **Z** and the weights **W**. When proposing a new row in **Z** the acceptance ratio depends only on $p(Y|.)$ and proposed number of new columns c* :

$$a = \min(1, \frac{p(Y|Z', W)}{p(Y|Z, W)} \frac{p(c^*|\alpha)}{p(c|\alpha)}) \tag{5}$$

. The old c is calculated as the number of columns that are all-zero after removing the row that we are resampling.

Probability $p(c|\alpha)$ is calculated as P(X=x) in Poisson distribution, and the $p(Y|Z, W)$ is calculated from by calculating the matrix **X** as described in the task with putting $p(y_{ij} = 0|Z, W) = 1 - sigmoid(x)$ elements of Y matrix that are observed to be 0.

If a random value drawn from $U(0, 1)$ is less than this acceptance ratio, then we accept the new number of columns.

The algorithm is implemented in methods propose_W() and propose_Z().

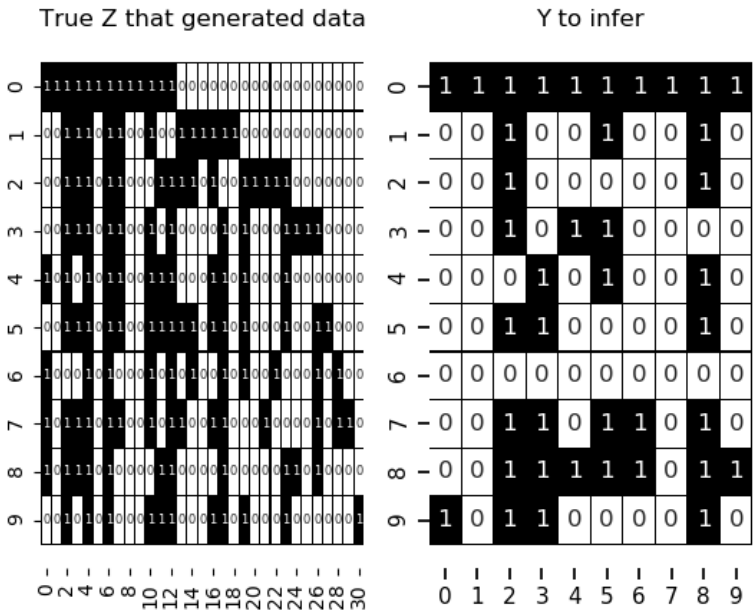
Question 16

Implement a generator or use the one that we will upload asap, generate synthetic data, and analyse it.

Generation of datasets yields matrices **X,W,Y,Z**. The Matrix **Z** is generated by $IBP(\alpha)$. It is a binary matrix with dimensions NxK,where N represents the number of objects and K is the number of features that we initially have. Z gives us information about features a certain object possesses.After we know the number of features, matrix **W** is randomly generated from a normal distribution (dimensions KxK). **X** and **Y** are then computed like it was described

in the model. In essence, the goal is to infer binary matrix **Y** which shows us connections between N objects (NxN dimensions). In Figure 18 the generated matrix **Z** from which the data was generated and matrix **Y** that represents connections between objects that we want to infer.

Figure 18: Generated data from $Z \sim IBP(\alpha)$



The code for generating data that was used was the one provided on the course Github repository.

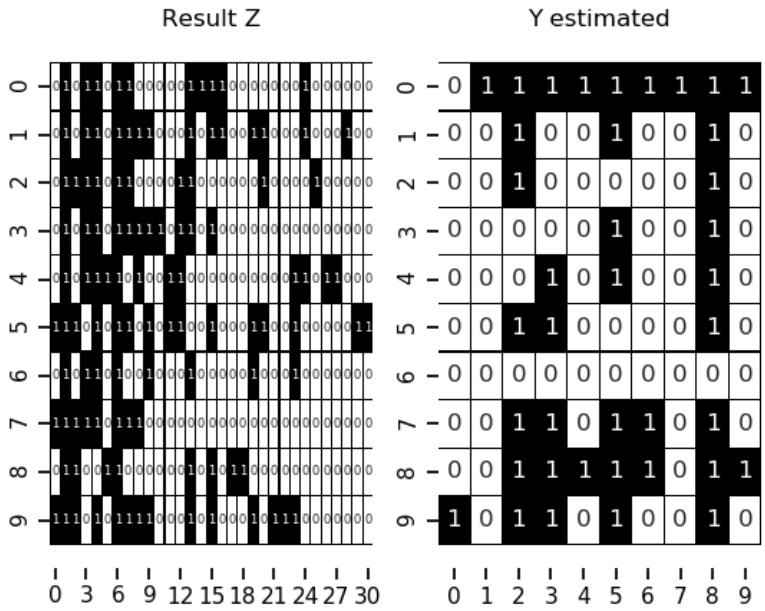
Question 17

Describe the performance.

After running the MCMC proposals for 20 iterations, where in each iteration we propose each row of matrix **Z** once, and after each row proposal we propose matrix **W**. Matrices **Z** and **Y** that we computed are shown in Figure 19. As we can see, estimated **Y** looks accurate when compared to the true **Y**. The accuracy of estimation is 0.95. The **Z** received from the sampling does not look alike to the **Z** that was used to generate the data. The reason for that is that in the MCMC we were resampling **Z** together with the weight matrix. In other words, even though this matrix is different from the true generating features, with the appropriate weight matrix which we also sample in every step, it predicts just

as well as the true feature matrix. While our latent feature approach is able to learn features that explain the data well, due to subtle interactions between sets of features and weights, the features themselves will not in general correspond to interpretable features as the true feature matrix.

Figure 19: Generated data from $Z \sim IBP(\alpha)$



References

[1] Fredrik Lindsten, Thomas B. Schön *Backward Simulation Methods for Monte Carlo Statistical Inference*. Foundations and Trends in Machine Learning Vol. 6, No. 1, 2013.

[2] Johan Dahlin, Thomas B. Schön *Getting Started with Particle Metropolis-Hastings for Inference in Nonlinear Dynamical Models*. Journal of Statistical Software, 2019.

[3] Kurt Miller, Michael I. Jordan, and Thomas L. Griffiths *Nonparametric Latent Feature Models for Link Prediction*. NIPS, pages 1276–1284, 2009.

[4] Thomas L. Griffiths and Zoubin Ghahramani *The Indian Buffet Process: An Introduction and Review*. Journal of Machine Learning Research, 12:1185–1224, 2011

[5] David Arthur Knowles *Bayesian non-parametric models and inference for sparse and hierarchical latent structure*. University of Cambridge, April 2012

[6] Kevin P. Murphy *Machine Learning : A probabilistic perspective* . MIT, 2012

4 Appendix

Listing 1: Task 2.1 Q1 + Q2

```
##############################################################################
# Function used to generate date according to instructions given in task 2.1
##############################################################################
generate.data <- function(N, alphabet, M, W, alpha.bg, alpha.mw){

  theta.bg <- rdirichlet(1, alpha.bg)
  theta.mw <- rdirichlet(W, alpha.mw) # for each magic word position

  data <- matrix(nrow = N, ncol = M)
  all.starts <- rep(NA, N)
  for(n in 1:N){

    generated.seq <- rep(NA, M)
    mw.start <- sample(1:(M - W + 1), 1)
    all.starts[n] <- mw.start
    mw.end <- mw.start + W - 1

    for(i in 1:M){

      if(i %in% mw.start:mw.end){ # it's a magic word part
        generated.seq[i] <- sample(alphabet, 1, prob = theta.mw[i - mw.start + 1, ])
      } else { # if it's background
        generated.seq[i] <- sample(alphabet, 1, prob = theta.bg)
      }
    }
    data[n, ] <- generated.seq
  }

  list(data = data, starts = all.starts)
}


##############################################################################
# Function used to extract magic words from sequences based on
# the current state - the current state represents starting positions
# of magic word for each of the sequences
##############################################################################
extract.magic.words <- function(data, current.state, N, W){
  magic.words <- matrix(nrow = N, ncol = W)
  end.positions <- current.state + W - 1
  for(i in 1:N){
    magic.words[i, ] <- as.matrix(data[i, current.state[i]:end.positions[i]])[1, ]
  }
  magic.words
}


##############################################################################
# Function used to extract background from sequences based on
# the current state - the current state represents starting positions
# of magic word for each of the sequences
##############################################################################
extract.backgrounds <- function(data, current.state, N, W){
  backgrounds <- matrix(nrow = N, ncol = M - W)
  end.positions <- current.state + W - 1
  for(i in 1:N){
    backgrounds[i, ] <- as.matrix(data[i, -c(current.state[i]:end.positions[i])])[1, ]
  }
  backgrounds
}


##############################################################################
# Function used to perform Gibbs sampling.
# It returns the chain, i.e. the sequence of states from all iterations
##############################################################################
gibbs.sampler <- function(my.data, alphabet, num.iterations, N, M, W, K, alpha.bg, alpha.mw){

  # the states are vectors of start positions
  initial.positions <- sample(1:(M - W + 1), N, replace = T)
  chain <- matrix(nrow = 0, ncol = N)
  chain <- rbind(chain, initial.positions)

  # do this only ONCE!
  magic.words <- extract.magic.words(my.data, initial.positions, N, W)
  backgrounds <- extract.backgrounds(my.data, initial.positions, N, W)


  for(it in 1:num.iterations){
    current.state <- chain[it, ]
    end.positions <- current.state + W - 1

    next.state <- rep(NA, N)
```

```
    # using current stat
    # precompute counts used for posterior

    ## counts for background: all characters except the magic words
    background.count.matrix <- matrix(nrow = N, ncol = K)

    for(n in 1:N){
      temp <- table(backgrounds[n, ])[alphabet] # use all, then modify in the loop
      temp[is.na(temp)] <- 0
      background.count.matrix[n, ] <- temp

    }

    for(n in 1:N){

      posterior.tilda <- posterior(my.data[n, ], current.state, n, magic.words, backgrounds,
                                   background.count.matrix)
      posterior.norm <- exp.normalize(posterior.tilda)
      new.start.for.n <- sample(1:(M - W + 1), size = 1, prob=posterior.norm)
      current.state[n] <- new.start.for.n # it's importat for the current iteration
      magic.words[n, ] <- my.data[n, new.start.for.n:(new.start.for.n+W-1)]
      backgrounds[n, ] <- my.data[n, -c(new.start.for.n:(new.start.for.n+W-1))]
      next.state[n] <- new.start.for.n
    }
    chain <- rbind(chain, next.state)

  }

  # chain represents sequence STATES for EACH iteration!
  # state = vector of starting positions for N sequences
  chain
}


##############################################################################################
# Function used to evaluate marginal likelihood of the background
##############################################################################################
bg.prob <- function(counts){
  part2 <- sum(lgamma(counts + alpha.bg) - loggamma.bg)
  log.p <- part1.bg.prob + part2
  log.p
}


##############################################################################################
# Function used to evaluate marginal likelihood of the magic words
##############################################################################################
mw.prob <- function(counts.position){
  part2 <- sum(lgamma(counts.position + alpha.mw) - loggamma.mw)
  log.p <- part1.mw.prob + part2
  log.p
}


##############################################################################################
# Normalize vector of values to [0, 1]
##############################################################################################
exp.normalize <- function(prob){
  prob <- exp(prob - max(prob))
  return(prob/sum(prob))
}


##############################################################################################
# Function used to evaluate posterior, i.e. full conditional
# probability p(r_n | R_{-n}, D)
##############################################################################################
posterior <- function(data, current.state, seq.id, magic.words, backgrounds, matrix.bg){
  prob.distribution <- rep(0, M - W + 1)

  # precompute for the magic word
  precomputed.mw <- matrix(nrow = W, ncol = K) # without seq
  mw.prob.no.first.last <- 0

  for(j in 1:W){
    temp <- table(magic.words[-seq.id, j])[alphabet] # index all seq. except seq.id
    temp[is.na(temp)] <- 0
    precomputed.mw[j, ] <- temp
    # for seq.id
    temp <- table(magic.words[seq.id, j])[alphabet] # separately seq.id
    temp[is.na(temp)] <- 0
    mw.prob.no.first.last <- mw.prob.no.first.last + mw.prob(temp + precomputed.mw[j, ])
  }

  precomputed.bg.sum <- matrix.bg[-seq.id, ] %>% apply(2, sum)
  magic.words[seq.id, ] <- data[1:W]

  oldTotal <- table(data)[alphabet]
  oldTotal[is.na(oldTotal)] <- 0

  for(new.position in 1:(M - W + 1)){
```

```
    end.position <- new.position + W - 1

    # calculate old first position
    temp.first <- table(magic.words[seq.id, 1])[alphabet]
    temp.first[is.na(temp.first)] <- 0
    # calculate old last position
    temp.last <- table(magic.words[seq.id, W])[alphabet]
    temp.last[is.na(temp.last)] <- 0
    # subtract probabilitiy values for old positions from ALL other sequences
    mw.prob.no.first.last <- mw.prob.no.first.last - mw.prob(temp.first + precomputed.mw[1, ]) -
      mw.prob(temp.last + precomputed.mw[W, ])
    # now you have posterior part without first and last position

    # permanently
    magic.words[seq.id, ] <- data[new.position:end.position] # modify one magic word
   # backgrounds[seq.id, ] <- data[-c(new.position:end.position)] # modify one background
    # permanently
    magic.count <- table(magic.words[seq.id, ])[alphabet]
    magic.count[is.na(magic.count)] <- 0

    counts.bg <- oldTotal - magic.count

    bg.prob.value <- bg.prob(counts.bg + precomputed.bg.sum)
    # try to replace old code
    # first and last position is changed
    temp.first <- table(magic.words[seq.id, 1])[alphabet]
    temp.first[is.na(temp.first)] <- 0

    temp.last <- table(magic.words[seq.id, W])[alphabet]
    temp.last[is.na(temp.last)] <- 0

    mw.prob.no.first.last <- mw.prob.no.first.last + mw.prob(temp.first + precomputed.mw[1, ]) +
      mw.prob(temp.last + precomputed.mw[W, ])

    prob.distribution[new.position] <- bg.prob.value + mw.prob.no.first.last
  }
  prob.distribution
}
```

Listing 2: Task 2.1 Q3

```
##############################################################################
# Function used to generate stochastic-volatility data
##############################################################################
generate_data <- function(fi, sigma, beta, total_len){
  library(tibble)

  results <- tibble(x = numeric(), y = numeric(), t = numeric())
  x <- rnorm(1, 0, sigma)
  y <- rnorm(1, 0, sqrt(beta^2 * exp(x)))
  results <- tibble::add_row(results, x = x, y = y, t = 1)
  for(i in 2:total_len){
    x <- rnorm(1, fi*x, sigma)
    y <- rnorm(1, 0, sqrt(beta^2 * exp(x)))
    results <- tibble::add_row(results, x = x, y = y, t = i)
  }
  return(results)
}
```

Listing 3: Task 2.2 Q4

```
##############################################################################
# Sequential importance sampling filter for stochastic volatility model
# N = number of particles
# T = number of time steps
##############################################################################
sis <- function(real.y, N, T){
  x.pf.sis <- matrix(0, nrow = T, ncol=N)
  w.pf.sis <- matrix(0, nrow = T, ncol=N)
  w.norm.sis <- matrix(0, nrow = T, ncol=N)

  x.pf.sis[1, ] <- rnorm(N, mean = 0, sd = 0.16) # Initial, t = 1

  w.pf.sis[1, ] <- dnorm(real.y[1], mean = 0, sd = sqrt(0.64**2 * exp(x.pf.sis[1, ]))) #
  w.norm.sis[1, ] <- w.pf.sis[1, ] / sum(w.pf.sis[1, ])

  x.means.sis <- rep(NA, T)
  x.means.sis[1] <- sum(w.norm.sis[1, ] * x.pf.sis[1,])

  for(t in 2:T){
    x.pf.sis[t, ] <- rnorm(N, mean = 1.00 * x.pf.sis[t-1, ], sd = 0.16)
    w.pf.sis[t, ] <- dnorm(real.y[t], mean = 0, sd = sqrt(0.64^2 * exp(x.pf.sis[t, ]))) * w.pf.sis[t-1, ]

    w.norm.sis[t, ] <- w.pf.sis[t, ] / sum(w.pf.sis[t, ])
```

```
    x.means.sis[t] <- sum(w.norm.sis[t, ] * x.pf.sis[t, ])
  }
  return(list(x.means = x.means.sis, wnorm = w.norm.sis))
}
```

Listing 4: Task 2.2 Q5 + Q6

```
##############################################################################
# Bootstrap particle filter
##############################################################################

bootstrap.filter <- function(real.y, sigma, beta, N, TIME, resampling.function){
  particles <- matrix(0, nrow = TIME, ncol = N)
  x.means <- matrix(0, nrow= TIME, ncol = 1)
  ancestor.indices <- matrix(1, nrow = TIME, ncol = N)
  weights <- matrix(1, nrow = TIME, ncol = N)
  weights.norm <- matrix(1, nrow = TIME, ncol = N)

  particles[1, ] <- rnorm(N, 0, sigma)
  ancestor.indices[1, ] <- 1:N

  weights[1, ] <- dnorm(real.y[1], mean = 0, sd = sqrt(beta^2 * exp(particles[1, ])), log = TIME)
  max.weight <- max(weights[1, ]) # max of log values
  weights[1, ] <- exp(weights[1, ] - max.weight)
  sum_log_weights <- sum(weights[1, ])
  weights.norm[1, ] <- weights[1, ] / sum_log_weights

  log.likelihood <- 0
  log.likelihood <- log.likelihood + max.weight + log(sum_log_weights) - log(N)

  x.means[1, 1] <- sum(weights.norm[1, ] * particles[1, ])

  for(t in 2:TIME){
    new.ancestors <- resampling.function(weights.norm[t-1, ])
    #particles[1:(t-1), ] <- particles[1:(t-1), new.ancestors]
    particles[t, ] <- rnorm(N, mean = 1.0*particles[t-1, new.ancestors], sigma) # using the most important particles

    weights[t, ] <- dnorm(real.y[t], 0, sqrt(beta**2 * exp(particles[t, ])), log = TRUE)
    max.weight <- max(weights[t, ]) # max of log values
    weights[t, ] <- exp(weights[t, ] - max.weight)
    sum_log_weights <- sum(weights[t, ])

    weights.norm[t, ] <- weights[t, ] / sum_log_weights

    x.means[t, 1] <- sum(weights.norm[t, ] * particles[t, ])

    log.likelihood <- log.likelihood + max.weight + log(sum_log_weights) - log(N)
  }

  ancestor.index <- sample(1:N, size=1, prob = weights.norm[TIME, ])
  x.hat.filtered <- particles[, ancestor.index]

  return(list(x.means= x.means, x.hat.filtered = x.hat.filtered, weights.norm = weights.norm, logl = log.likelihood))
}
```

Listing 5: Task 2.2 Q7

```
library(tibble)
library(ggplot2)
library(magrittr)
library(RColorBrewer)

source("bootstrap_filter.R")
source("resampling_methods.R")
source("data_generator.R")

set.seed(2723)
my.data <- generate_data(1.00, 0.16, 0.64, 100)
bpf.multinomial <- bootstrap.filter(my.data$y, 0.16, 0.64, 200, 100, multinomial.resampling)

# visualize
tibble(t = 1:100, value = bpf.multinomial$x.means) %>%
  ggplot(aes(x = t, y = value)) +
  geom_line(aes(color = "a")) +
  geom_line(data = tibble(t = 1:100, x = my.data$x), aes(x = t, y = x, color = "b"),
            alpha = 0.4) +
  ggtitle("Predicted vs real values") +
  scale_color_manual(name = NULL, breaks = c("a", "b"),
                     labels = c("BPF", "real data"), values = c("red", "blue"))


# point estimate
bpf.multinomial$x.means[100]
my.data$x[100]

# MSE
```

```
mse.tibble.bpf.multinomial <- tibble(N = as.numeric(), mse = as.numeric())
for(N in seq(10,1500, 5)){
  bpf.multinomial <- bootstrap.filter(my.data$y, 0.16, 0.64, N, 100, multinomial.resampling)
  mse.tibble.bpf.multinomial <- add_row(mse.tibble.bpf.multinomial, N = N, mse = (bpf.multinomial$x.means[100] - my.data$x[100])^2)
}

ggplot(mse.tibble.bpf.multinomial, aes(x = N, y = mse)) +
  geom_line() + ylab("MSE")

# visualise weights
tibble(value = as.numeric(bpf.multinomial$weights.norm[100, ])) %>% ggplot(aes(x = value)) +
  geom_histogram(bins = 30)
ggsave("bpf_multinomial_histogram.png")




##############
bpf.stratified <- bootstrap.filter(my.data$y, 0.16, 0.64, 200, 100, stratified.resampling)
tibble(t = 1:100, value = bpf.stratified$x.means) %>%
  ggplot(aes(x = t, y = value)) +
  geom_line(aes(color = "a")) +
  geom_line(data = tibble(t = 1:100, x = my.data$x), aes(x = t, y = x, color = "b"),
            alpha = 0.4) +
  ggtitle("Predicted vs real values") +
  scale_color_manual(name = NULL, breaks = c("a", "b"),
                     labels = c("BPF", "real data"), values = c("red", "blue"))



#MSE
mse.tibble.bpf.stratified <- tibble(N = as.numeric(), mse = as.numeric())
for(N in seq(10,1500, 5)){
  bpf.stratified <- bootstrap.filter(my.data$y, 0.16, 0.64, N, 100, stratified.resampling)
  mse.tibble.bpf.stratified <- add_row(mse.tibble.bpf.stratified, N = N, mse = (bpf.stratified$x.means[100] - my.data$x[100])^2)
}

ggplot(mse.tibble.bpf.stratified, aes(x = N, y = mse)) +
  geom_line() + ylab("MSE")

# weights histogram
tibble(value = as.numeric(bpf.stratified$weights.norm[100, ])) %>% ggplot(aes(x = value)) +
  geom_histogram(bins = 5)
  ggsave("bpf_stratified_histogram.png")


# Compare all three methods
sis.filter <- sis(my.data$y, 200, 100)
tibble(value = as.numeric(sis.filter$wnorm[100, ])) %>% ggplot(aes(x = value)) +
  geom_histogram(bins = 30)
ggsave("sis_histogram.png")

ggplot(data = tibble(t = 1:100, x= bpf.multinomial$x.means), aes(x = t, y = x, color = "c")) +
  geom_line(aes(color = "b")) +
  geom_line(data = tibble(t = 1:100, x = my.data$x), aes(x = t, y = x, color = "a") , linetype = "dashed") +
  geom_line(data = tibble(t = 1:100, x = sis.filter$x.means), aes(t, x, color = "b")) +
  geom_line(data = tibble(t = 1:100, x = bpf.stratified$x.means), aes(t, x)) +
  geom_line(aes(color = "d")) +
  scale_color_manual(name = NULL,
    breaks = c("a", "b", "c", "d"),
                       labels = c("Real data", "SIS", "BPF multinomial", "BPF stratified"),
                       values = c("#666666", "#D95F02", "#66A61E", "#7570B3")) +
  theme(legend.text = element_text(size = 18))
ggsave("comparison_filters.png", units = c("cm"), width = 30)
```

Listing 6: Task 2.3

```
# TASK 2.3 Question 9
loglikelihood <- function(real.y, sigma, beta, N, T, resampling.function){
  all.results <- c()
  for(i in 1:10){
    all.results <- c(all.results, bootstrap.filter(real.y, sigma, beta, N, T, resampling.function)$logl)
  }
  return(mean(all.results))
}


# TASK 2.3 Question 10
pmh <- function(y.values, sigma, beta, Nparticles, TIME, Niteration, step.size, resampling.function){

  theta <- matrix(0, nrow = 2, ncol = Niteration)
  theta.proposed <- matrix(0, nrow = 2, ncol = Niteration)

  log.likelihood <- matrix(0, nrow = 1, ncol = Niteration)
  log.likelihood.proposed <- matrix(0, nrow = 1, ncol = Niteration)
  # same for acceptance, it is only kept for time = t
```

```
  proposed.accepted <- matrix(0, nrow = 1, ncol = Niteration)

  # initial values
  theta[1, 1] <- sigma
  theta[2, 1] <- beta

  results0 <- bootstrap.filter(y.values, sigma=sqrt(theta[1, 1]), beta=sqrt(theta[2, 1]), Nparticles, TIME, resampling.function)
  log.likelihood[1, 1] <- results0$logl


  for(k in 2:Niteration){

    theta.proposed[1,k]<- max(0.0001, rnorm(1, theta[1,k-1], step.size[1]))
    theta.proposed[2,k]<- max(0.01, rnorm(1, theta[2,k-1], step.size[2]))

    results <- bootstrap.filter(y.values, sigma=sqrt(theta.proposed[1, k]),
                                beta=sqrt(theta.proposed[2, k]), Nparticles, TIME, resampling.function)
    log.likelihood.proposed[1, k] <- results$logl


    prior.sigma <- nimble::dinvgamma(theta.proposed[1, k], shape = 0.01, scale = 0.01, log = TRUE)
    diff.sigma <- prior.sigma - nimble::dinvgamma(theta[1, k-1], shape = 0.01, scale = 0.01, log = TRUE)

    prior.beta <- nimble::dinvgamma(theta.proposed[2, k], shape = 0.01, scale = 0.01, log = TRUE)
    diff.beta <- prior.beta - nimble::dinvgamma(theta[2, k-1], shape = 0.01, scale = 0.01, log = TRUE)
    prior.diff.sum <- diff.sigma + diff.beta

    likelihood.dif <- log.likelihood.proposed[1, k] - log.likelihood[1, k-1]
    accept.probability <- exp(prior.diff.sum + likelihood.dif)

    uniform <- runif(1)
    if(uniform <= accept.probability){ # accept new parameters
      theta[1:2, k] <- theta.proposed[1:2, k]
      log.likelihood[1, k] <- log.likelihood.proposed[1, k]
      proposed.accepted[1, k] <- 1
    } else {
      theta[1:2, k] <- theta[1:2, k-1] # note! it is not PROPOSED theta
      log.likelihood[1, k] <- log.likelihood[1, k-1]
      proposed.accepted[1, k] <- 0
    }

    if(k %% 50 == 0){

     cat(sprintf("Current posterior mean for sigma and beta: %f %f \n",mean(theta[1, 1:k]),mean(theta[2, 1:k])))

    }
  }
  # theta is matrix
  return(list(theta = theta, proposed.accepted = proposed.accepted))

}
```

Listing 7: Task 2.3 graphs

```
source("util.R")
source("../2.2/bootstrap_filter.R")
source("../2.2/resampling_methods.R")
source("../2.2/data_generator.R")

library(tibble)
library(dplyr)
library(ggplot2)
library(magrittr)
library(purrr)
library(invgamma)

search.grid <- tibble(
    sigma = numeric(),
    beta = numeric(),
    logl = numeric()
 )

 for(s in seq(0.001,2,by=0.08)){
    for(b in seq(0.001,2,by=0.08)){
    search.grid <- add_row(search.grid, sigma = s, beta = b)
    }
 }

 my.data <- generate_data(1.00, 0.16, 0.64, 100)

 for(row in 1:nrow(search.grid)){
    search.grid[row,"logl"] <- loglikelihood(real.y = my.data$y,
                                             sigma = search.grid[row,"sigma"] %>% pull,
                                             beta = search.grid[row,"beta"] %>% pull,
                                             200, 100, multinomial.resampling)
 }

 # check for highest values
```

```
search.grid %>% arrange(-logl) %>% top_n(20)


mlogl <- search.grid %>% mutate(logl = ifelse(logl > -250, logl, NA)) %>% filter(!is.na(logl)) %>%
  summarize(medianlogl = mean(logl)) %>% pull
search.grid %>% mutate(logl = ifelse(logl > -100, logl, NA)) %>%
  ggplot(aes(x = sigma, y = beta, fill = logl)) +
  geom_tile() +scale_fill_gradient2(midpoint = mlogl, low = "blue", mid = "white",
                                    high = "red", space = "Lab") +
  xlab(expression(sigma)) +
  ylab(expression(beta)) +
  theme(axis.title.y = element_text(angle = 0))

# Question 9

set.seed(100)
runif(1)
runif(1)
my.data <- generate_data(1.00, 0.16, 0.64, 100)

varyT <- tibble(
  Tvalue = numeric(),
  logl = numeric()
)
# Fix N = 100
for(t in c(25, 50, 60, 70, 80, 100)){
  for(i in 1:100){ # to get enough samples for each T
    varyT <- add_row(varyT, Tvalue = t, logl = bootstrap.filter(my.data$y, 0.16,0.64, 200, t, multinomial.resampling)$logl)
  }
}

ggplot(varyT, aes(x = Tvalue, group = Tvalue, y = logl)) +
  geom_boxplot() + ylab("log-likelihood") + xlab("t")
ggsave("boxplot_vary_t.png")

varyN <- tibble(
  Nvalue = numeric(),
  logl = numeric()
)
# Fix N = 100
for(n in c(25, 50, 100, 200, 300, 500)){
   for(i in 1:100){ # to get enough samples for each T
    varyN <- add_row(varyN, Nvalue = n, logl = bootstrap.filter(my.data$y, 0.16,0.64, n, 100, multinomial.resampling)$logl)
}
}

ggplot(varyN, aes(x = Nvalue, group = Nvalue, y = logl)) +
  geom_boxplot() + ylab("log-likelihood") + xlab("N")
ggsave("boxplot_vary_n.png")

##############################################################################
# PARTICLE METROPOLIS HASTINGS
##############################################################################
# 2000, 0.01, 0.2 c(0.001, 0.01) works fine for BETA
my.data <- generate_data(1.00, 0.16, 0.64, 100)
Niteration <- 10000
burnIn <- 5000
pmh.results <- pmh(my.data$y, 0.1, 0.1, 200, 100, Niteration, c(0.01, 0.2), multinomial.resampling)

PMH.posterior.sigma2 <- tibble(sigma2 = pmh.results$theta[1,burnIn:Niteration])
PMH.posterior.beta2 <- tibble(beta2 = pmh.results$theta[2,burnIn:Niteration])

acceptance.ratio <- sum(pmh.results$proposed.accepted)/length(pmh.results$proposed.accepted)
# visualize distributions

# SIGMA^2
ggplot(PMH.posterior.sigma2, aes(x = sigma2)) +
  geom_histogram(bins = 30, aes(y = ..density..)) +
  geom_density(alpha = 0.2, fill = "lightblue") +
  geom_vline(xintercept = mean(PMH.posterior.sigma2$sigma2), size = 1.2, linetype = 3, color = "blue") +
  xlab(expression(sigma^{2})) +
  ylab("") +
  scale_y_discrete(breaks = NULL) +
  annotate('text', x = 0.06, y = 30,
           label = paste("bar(sigma^2)==~",round(mean(PMH.posterior.sigma2$sigma2), 4)),
           parse = TRUE,size=10)

ggsave("./sigma2.png")

# BETA^2
ggplot(PMH.posterior.beta2, aes(x = beta2)) +
  geom_histogram(bins = 30, aes(y = ..density..)) +
  geom_density(alpha = 0.2, fill = "lightblue") +
  geom_vline(xintercept = mean(PMH.posterior.beta2$beta2), size = 1.2, linetype = 3, color = "blue") +
  xlab(expression(beta^{2})) +
  ylab("") +
  annotate('text', x = 1, y =2,
           label = paste("bar(beta^2)==~",round(mean(PMH.posterior.beta2$beta2), 4)),
           parse = TRUE,size=10) +
  scale_y_continuous(breaks = NULL)
```

```
ggsave("./beta2.png")

# line plot
tibble(sigma2 = pmh.results$theta[2,]) %>% mutate(iterations = 1:Niteration) %>% ggplot(aes(x = iterations,
                                                                                           y = sigma2)) +
  geom_line()
```

Listing 8: Task 2.4 Q10

```
# TASK 2.4
################################################################################
# Function to perform conditional SMC
################################################################################
csmc <- function(x.values, y.values, sigma, beta, num.particles, TI){

  # create data structures
  particles <- matrix(0, nrow = TI, ncol = num.particles)
  weights <- matrix(0, nrow = TI, ncol = num.particles)
  weights.norm <- matrix(0, nrow = TI , ncol = num.particles)
  ancestor.indices <- matrix(0, nrow = TI, ncol = num.particles)

  particles[1, ] <- rnorm(num.particles, 0, sigma)
  particles[1, num.particles] <- x.values[1] # set the retained particle
  ancestor.indices[1, ] <- 1:num.particles

  weights[1, ] <- dnorm(y.values[1], 0, sd=sqrt(beta^2 * exp(particles[1, ])), log = T)
  max.weight <- max(weights[1, ])
  weights[1, ] <- exp(weights[1, ] - max.weight)
  weights.norm[1, ] <- weights[1, ] / sum(weights[1, ])
  log.likelihood <- 0
  log.likelihood <- log.likelihood + max.weight + log(sum(weights[1, ])) - log(num.particles)

  for(t in 2:TI){

    new.ancestors <- multinomial.resampling(weights.norm[t-1, ])
    new.ancestors[num.particles] <- num.particles

    ancestor.indices[t,] <- ancestor.indices[t-1, new.ancestors]

    particles[t, ] <- rnorm(num.particles, particles[t-1, ancestor.indices[t,]], sigma)
    particles[t, num.particles] <- x.values[t] # replace last one
    weights[t, ] <- dnorm(y.values[t], 0, sd = sqrt(beta^2 * exp(particles[t, ])), log = T)

    max.weight <- max(weights[t, ])
    weights[t, ] <- exp(weights[t, ] - max.weight)
    weights.norm[t, ] <- weights[t, ] / sum(weights[t, ])

    log.likelihood <- log.likelihood + max.weight + log(sum(weights[t, ])) - log(num.particles)
  }

    index <- sample(1:num.particles, size = 1, prob = weights.norm[TI, ])
    x.star <- rep(0, TI)
    x.star[TI] <- particles[TI, index]

    for(t in TI:2){ # trace the ancestral path of particle index
      index <- ancestor.indices[t, index]
      x.star[t-1] <- particles[t-1, index]

    }
    list(x.star = x.star, ancestors = ancestor.indices, logl = log.likelihood)

}


################################################################################
# Function to perform Gibbs sampling
################################################################################
particle.gibbs <- function(y.values, sigma2, beta2, TI, num.particles, num.iteration){
  current.sigma2 <- sigma2
  current.beta2 <- beta2

  # set x values arbitrarily
  bf <- bootstrap.filter(y.values, sqrt(sigma2), sqrt(beta2), num.particles, TI, multinomial.resampling)
  current.x <- bf$x.means


  csmc.result <- csmc(current.x, y.values, sqrt(current.sigma2), sqrt(current.beta2),
                      num.particles, TI)
  current.x <- csmc.result$x.star


  sigma2seq <- tibble(sigma2 = numeric())
  beta2seq <- tibble(beta2 = numeric())

  sigma2seq <- add_row(sigma2seq, sigma2 = current.sigma2)
  beta2seq<- add_row(beta2seq, beta2 = current.beta2)

  log.likelihood.iters <- rep(NA, num.iteration)
```

```
  log.likelihood.iters[1] <- bf$logl
  for(k in 2:num.iteration){

    current.sigma2 <- nimble::rinvgamma(1, shape = 0.01 + TI/2,
                                        scale = 0.01 + 0.5 * sum( ( current.x[2:TI] - current.x[1:(TI-1)] )^2))

    sigma2seq <- add_row(sigma2seq, sigma2 = current.sigma2)

    csmc.result <- csmc(current.x, y.values, sqrt(current.sigma2), sqrt(current.beta2),
                                             num.particles, TI)

    current.x <- csmc.result$x.star
    current.beta2 <- nimble::rinvgamma(1, shape = 0.01 + TI/2,
                                          scale = 0.01 + 0.5 * sum(exp(-current.x)*(y.values^2)))
    beta2seq <- add_row(beta2seq, beta2 = current.beta2)
    log.likelihood.iters[k] <- csmc.result$logl
    if(k %% 150 == 0){
      cat(sprintf("Log-likelihood = %.4f \n", csmc.result$logl))
    }
  }

  list(sigma.density = sigma2seq$sigma2, beta.density = beta2seq$beta2, logl = log.likelihood.iters)

}
```

Listing 9: Task 2.4 graphs

```
source("../2.2/data_generator.R")
source("../2.2/resampling_methods.R")
source("util.R")
source("../2.2/bootstrap_filter.R")

library(tidyverse)

set.seed(12793129)
my.data <- generate_data(1.00, 0.16, 0.64, 100)

set.seed(5055)
iter <- 20000
results.gibbs <- particle.gibbs(my.data$y, 0.1, 0.1, 100, 100, iter)
burn.in <- 5000
plot(results.gibbs$sigma.density)
plot(results.gibbs$beta.density)



PG.posterior.sigma2 <- tibble(sigma2 = results.gibbs$sigma.density[burn.in:20000])


ggplot(PG.posterior.sigma2, aes(x = sigma2)) +
  geom_histogram(bins = 30, aes(y = ..density..)) +
  geom_density(alpha = 0.2, fill = "lightblue") +
  geom_vline(xintercept = mean(PG.posterior.sigma2$sigma2), size = 1.2, linetype = 3, color = "blue") +
  xlab(expression(sigma^{2})) +
  ylab("") +
  scale_y_discrete(breaks = NULL) +
  #ggtitle( expression(paste("Distribution of ", sigma^{2}))) +
  annotate('text', x = 0.06, y = 30,
           label = paste("bar(sigma^2)==~",round(mean(PG.posterior.sigma2$sigma2), 4)),
           parse = TRUE,size=10)
ggsave("./pg-sigma2.png")

PG.posterior.beta2 <- tibble(beta2 = results.gibbs$beta.density[burn.in:20000])
ggplot(PG.posterior.beta2, aes(x = beta2)) +
  geom_histogram(bins = 30, aes(y = ..density..)) +
  geom_density(alpha = 0.2, fill = "lightblue") +
  geom_vline(xintercept = mean(PG.posterior.beta2$beta2), size = 1.2, linetype = 3, color = "blue") +
  xlab(expression(beta^{2})) +
  ylab("") +
  #ggtitle( expression(paste("Distribution of ", sigma^{2}))) +
  annotate('text', x = 0.85, y = 2,
           label = paste("bar(sigma^2)==~",round(mean(PG.posterior.beta2$beta2), 4)),
           parse = TRUE,size=10) +
  scale_y_continuous(breaks = NULL)
ggsave("./pg-beta2.png")

both <- tibble(x = PG.posterior.sigma2$sigma2, y = PG.posterior.beta2$beta2, t = burn.in:20000)

ggplot(both, aes(x = x, y = y, color = t)) +
  geom_path() +
  xlab(expression(sigma^{2})) +
  ylab(expression(beta^{2})) +
  geom_vline(xintercept = 0.16**2, color = "red", linetype = 2, size = 1) +
  geom_hline(yintercept = 0.64**2, color = "red", linetype = 2, size = 1) +
  scale_color_continuous(name = "iteration") +
  theme(axis.title.y = element_text(angle = 0))
ggsave("./pg-both.png")
```

```
tibble(logl = results.gibbs$logl, k = 1:iter) %>% ggplot(aes(k, logl)) +
  geom_line()
```

Listing 10: Task 3.1 Q12

```
def generate_data(alpha, n, d0=1000, a0=1, a1=1):
    # pi ~ GEM(alpha)
    # Zn ~ pi
    # theta_k ~ H0= Beta(a0,a1)
    # d_n ~ Poisson(d0)
    # b_n ~ Bin(d_n,theta_Zn)

    np.random.seed(256)
    pi = gem(alpha)
    # generating component labels
    # Zn = np.sort(list(map(lambda x : np.argmax(x), np.random.dirichlet(pi,n))))
    Zn = np.sort(list(map(lambda x: np.argmax(x), np.random.multinomial(1, pi, n))))
    # generating parameters for binomial(probabilities)
    thetas = np.random.beta(a0, a1, len(pi))
    # generating dn
    dn = get_dn(d0, n)

    # generating bn
    cnt = Counter(Zn)

    bn = np.array([])

    """since dn generation is independent of bn sampling from binomial,
    the component labels were sorted WLOG"""
    start = 0

    for i in range(len(pi)):
        if (cnt[i] != 0):
            # bn.append(np.random.binomial(dn[:cnt[i]],thetas[i],cnt[i]))

            bn = np.concatenate((bn, np.random.binomial(dn[start:cnt[i] + start], thetas[i], cnt[i])))
        start += cnt[i]

    return dn, bn, Zn

def gem(alpha):
    # stick breaking construction
    pi = []
    i = 0
    remaining = 1.0 # 1.0 stick
    i = 0

    while (remaining > 0.00001):
        beta_k = np.random.beta(1, alpha, 1)[0]
        pi.append(beta_k * remaining)
        remaining *= (1 - beta_k)
    pi.append(remaining)

    return pi
```

Listing 11: Task 3.1 Q13 + Rand index

```
class DPMM():
    def __init__(self, data, alpha,a=1,b=1):

        self.a = a
        self.b= b
        self.N = len(data)
        self.alpha = alpha
        self.z = None
        self.data = data
        self.initialize()
        self.ns =None
        self.rs = None

        self.compute_clusters()

    def compute_clusters(self, data_index=None):
        # if data_index is provided, it is the index of the element we need to remove from the calculations

        # TODO inicijaliziraj klustere
        if data_index is not None:
            self.z[data_index, :] = 0
            # remove column if the cluster is empty
            column_count = np.sum(self.z, axis=0)
            # we have the counts here, now we can
            non_zero_cols = np.where(column_count > 0)[0]
            self.z = self.z[:, non_zero_cols]
            self.n_classes = self.z.shape[1]

        self.counts = np.sum(self.z, axis=0)
```

```
    self.compute_parameters()

def compute_parameters(self):
    self.ns = np.zeros((self.n_classes))
    self.rs = np.zeros((self.n_classes))

    for class_index in range(self.n_classes):


        index_of_classes = self.z[:,class_index] >0
        self.ns[class_index] = np.sum(self.data[index_of_classes,1])
        self.rs[class_index] = np.sum(self.data[index_of_classes,0])


def initialize(self):

    z = np.zeros((self.N, 1))
    z[0] = 1
    for n in range(self.N):
        k = z.shape[1]
        prob = np.zeros(k + 1)
        prob[0:k] = np.sum(z, axis=0)
        prob[k] = self.alpha
        prob = prob / sum(prob)
        class_ind = np.random.choice(list(range(k + 1)), p=prob)
        if class_ind == k:
            # we chose a new class
            col = np.zeros((self.N, 1))
            z = np.append(z, col, axis=1)
        z[n, class_ind] = 1
    self.z = z
    self.n_classes = z.shape[1]

def collapsed_gibbs_sampling(self, n_iter=100):
    # TODO collapsed gibbs sampling
    for i in range(n_iter):
        self.sample()

def sample(self):

    for i in range(self.N):
        self.compute_clusters(data_index=i)


        prob = self.compute_pred_prob(i)


        cls_assignment = np.random.choice(list(range(len(prob))), p=prob)
        z = self.z
        if cls_assignment == len(prob) - 1:
            """new class """

            col = np.zeros((self.N, 1))
            z = np.append(z, col, axis=1)

        z[i, cls_assignment] = 1
        # remove zero columns/ empty cluster
        column_counts = np.sum(z, axis=0)
        nzc = np.where(column_counts > 0)[0]
        z = z[:, nzc]
        self.z = z
        self.nClass = z.shape[1]


    self.compute_clusters()
    return

def compute_pred_prob(self, data_index):
    predictive = np.zeros(self.n_classes + 1)
    # adding one class for a probability that the sample n belongs to
    # a new class

    for class_index in range(self.n_classes):

        n = self.ns[class_index]
        r = self.rs[class_index]

        betabinom_log = betabinom.logpmf(self.data[data_index][0],self.data[data_index][1],1+r,self.alpha+n-r)
        predictive[class_index] = betabinom_log + np.log(self.counts[class_index]/(self.alpha+self.N-1))
    ##predictive for a new class

    betabinom_log = betabinom.logpmf(self.data[data_index][0], self.data[data_index][1], 1, self.alpha)

    predictive[self.n_classes] = betabinom_log +np.log(self.alpha/(self.alpha+self.N-1))

    # computing the probability
    prob = predictive - max(predictive)
    prob = np.exp(prob)

    prob = prob / np.sum(prob)
```

```
        return prob
def calculate_rand_index(actual,predicted) :
    """calculating rand index for accuracy of clustering
        at pairs level"""
    a = 0
    b = 0
    all_pairs = special.binom(len(actual), 2)
    for i in range(len(actual) - 1):
        for j in range(i + 1, len(actual)):
            if (predicted[i] == predicted[j] and actual[i] == actual[j]):
                a += 1
            elif (predicted[i] != predicted[j] and actual[i] != actual[j]):
                b += 1

    rand_index = (a + b) / all_pairs
    return rand_index
```

Listing 12: Task 3.1 Q14

```
def perf_alpha(alphas,a0=1,a1=1,iterations=10,data_size=30,d0=1000) :
    performance_measures = []
    for alpha in alphas:
        dn, bn, label_clusters = generate_data(alpha, data_size, d0=d0,a0=a0,a1=a1)
        dataset = np.array(list(zip(bn, dn)))
        # print(data)

        dpmm = DPMM(dataset, 1, 1)
        dpmm.collapsed_gibbs_sampling(n_iter=iterations)

        z = dpmm.get_z()
        z = np.argmax(z, axis=1)
        performance_measures.append(calculate_rand_index(label_clusters, z))

    return performance_measures

def perf_data_size(sizes,alpha=1,a0=1,a1=1,iterations=10,d0=1000):
    performance_measures = []
    for n in sizes:
        dn, bn, label_clusters = generate_data(alpha, n, d0=d0,a0=a0,a1=a1)
        dataset = np.array(list(zip(bn, dn)))
        # print(data)

        dpmm = DPMM(dataset, 1, 1)
        dpmm.collapsed_gibbs_sampling(n_iter=iterations)

        z = dpmm.get_z()
        z = np.argmax(z, axis=1)
        performance_measures.append(calculate_rand_index(label_clusters, z))

    return performance_measures
```

Listing 13: Task 3.1 Graphs

```
def scatter_plot(dn,bn,label_clusters,name='',x='',y='',s=4) :
    LABEL_COLOR_MAP = {0: 'r', 1: 'b', 2: 'g', 3: 'y', 4: 'c', 5: 'm', 6: 'y', 7: 'k'}
    marker_style = ["+", "*",'o','x','s']
    label_color = [LABEL_COLOR_MAP[l % 8] for l in label_clusters]

    plt.title(name)
    plt.xlabel('dn ~ Poisson(d0)')
    plt.ylabel('bn ~ Bin(dn,theta_k)')


    plt.scatter(dn, bn, c=label_color, s=s)
    return
    #plt.show()


    #performance
    alphas = list(range(1,20))
    performance_measures= perf_alpha(alphas,data_size=70,iterations=5)

    print("Performance measure when alpha is increasing ",performance_measures)

    plt.subplot(121)
    plt.plot(alphas,performance_measures)
    plt.title('Alpha increasing')
    plt.xlabel('alpha')
    plt.ylabel('Rand index')



    ns = list(range(30,2031,100))
```

```
    performance_measures = perf_data_size(ns,alpha=5,iterations=5)
    print("Performance measure when N is increasing ", performance_measures)
    plt.subplot(122)
    plt.plot(ns, performance_measures)
    plt.title('data increasing')
    plt.xlabel('data size')
    plt.ylabel('Rand index')
    plt.savefig('Performance_dpmm.png')
    plt.show()
```

Listing 14: Task 3.2 Q15

```
def y_given_rest(Z, W,Y_true):

    #calculating p(Y|.)
    #p(y_ij = 1 |.) = sigmoid(zwz)
    #p(y_ij=0|.) = 1-sigmoid(zwz)
    Y = Y_true[0]
    X = np.dot(Z, W)
    X = np.dot(X, np.transpose(Z))

    probabilities = []

    for i in range(len(Y)):
        for j in range(len(Y[0])):
            X[i][j] = sigmoid(X[i][j])
            if(Y[i][j]==1) :
                probabilities.append(X[i][j])
            else :
                probabilities.append(1-X[i][j])

    return np.array(probabilities)


def propose_Z (Z,W,Y,row,alpha=10) :

        N_objects = len(Z)
        N_features = len(Z[0])

        #count non-zero columns
        m = np.sum(Z,axis=0)


        #Z_new = np.copy(Z)

        #for every entity (object) propose a new value (row in Z)
        Z_old = np.copy(Z)
        W_old = np.copy(W)
        m = np.sum(Z, axis=0)

        i = row
        empty_columns = []
        for k in range(len(Z[0])):

            m_ik = m[k]-Z[i][k]
            if(m_ik > 0 ) :
                #Without the N_th customer, the column is not all-zero
                #Gibbs sample z_ik
                prob = m_ik/N_objects
                random = np.random.uniform(0,1)
                if(random <= prob ) :
                    #with probability prob we are choosing this dish
                    Z[i][k]=1
                else :
                    #probability 1-prob we are not choosing this dish
                    Z[i][k]=0
            else :
                empty_columns.append(k-len(empty_columns))
                #the column is all zero
                #Z = np.delete(Z,k,1)

        for column in empty_columns:
            Z = np.delete(Z,column,1)
            W = np.delete(W,column,0)
            W = np.delete(W,column,1)
        N = len(Z)
        K = len(Z[0])
        new_dishes = np.random.poisson(alpha/N_objects)

        if (new_dishes >0) :

            Z_new = np.zeros((N, K + new_dishes))
            Z_new[:, :-new_dishes] = Z

            Z_new[i,K:K+new_dishes] = 1

            #print(new_dishes)
            W_new = np.zeros((K+new_dishes,K+new_dishes))
```

```
            W_new[:-new_dishes,:-new_dishes]=W
            #W = np.copy(W_new)

            for k1 in range(K,K+new_dishes) :
                for k2 in range(K,K+new_dishes):

                    W_new[k1][k2] = np.random.normal(0,1)
            Z=np.copy(Z_new)
            W=np.copy(W_new)
        elif(np.array_equal(Z_old,Z)) :
            #The customer picked exactly the same dishes
            return (Z_old,W_old)

        singleton_t1 = len(empty_columns) #singleton values were the ones chosen by poisson(alpha/n)
        singleton_t2 = new_dishes

        p_singleton_t1 = stats.poisson.pmf(singleton_t1,alpha/N)
        p_singleton_t2 = stats.poisson.pmf(singleton_t2,alpha/N)

        p_y_old = y_given_rest(Z_old,W_old,Y)

        Z = np.array(Z)
        W= np.array(W)

        p_y_new = y_given_rest(Z,W,Y)


        al = np.prod(p_y_new/p_y_old)
        ap= p_singleton_t2/p_singleton_t1

        """calculate acc prob,
        evade underflow"""
        a= al*ap
        a = min(1,a)


        u = np.random.uniform(0,1)
        if (u<a) :
            #Z_new = z_proposed
            Z = np.copy(Z)
            W = np.copy(W)

        else :
            #z_new = z_prior
            Z = np.copy(Z_old)
            W = np.copy(W_old)

        return(Z,W)

def propose_W (Z,W,Y,row,sigma=1) :


    #print(W.shape,Z.shape)
    W_old = np.copy(W)

    for i in range(len(W)) :
        for j in range(len(W[0])):

            w_old = W[j][i]
            w_proposed = np.random.normal(loc=w_old,scale=sigma)
            W[j][i] = w_proposed
            p_w_old = stats.norm.pdf(w_old,loc=0,scale=sigma)
            p_w_new = stats.norm.pdf(w_proposed,loc=0,scale=sigma)
            p_wn_w = stats.norm.pdf(w_proposed, loc=w_old,scale=sigma)
            p_w_wn = stats.norm.pdf(w_old, loc=w_proposed,scale=sigma)


            new_contr =1
            old_contr =1

            Z=np.array(Z)
            W=np.array(W)

            p_y_new= y_given_rest(Z,W,Y)
            p_y_old = y_given_rest(Z,W_old,Y)

            a= np.prod(p_y_new/p_y_old) * (p_w_old/p_w_new)*(p_wn_w/p_w_wn)
            a=min(1,a)
            u = np.random.uniform(0,1)

            if(u<= a) :
                W = np.copy(W)

            else :
                W = np.copy(W_old)


    return (Z,W)
```

Listing 15: Task 3.2 Q16

```
# import IBPDataGenerator
#
# # generation of a graph Z ~ IBP(alpha)
# N = 10
# alpha = 10.0
# Z = IBPDataGenerator.IBP_generator(N, alpha, saveOpt=True, displayOpt=True, filename='myIBP_Z.csv')
#
# # generation of a weight matrix W and of a link probability matrix X for Graph Z
# weight_sigma = 1
# [X,W] = IBPDataGenerator.LinkProbabilityMatrix_generator(Z, weight_sigma, displayOpt=True, saveOpt=True, filenameX='myIBP_X.csv', filenameW='my
#
# # generation of a link matrix Y for a given link probability matrix X
# Y = IBPDataGenerator.LinkMatrix_generator(X, displayOpt=True, saveOpt=True, filename='myIBP_Y.csv')
#
from math import exp
from numpy.random import normal
from numpy.random import seed, poisson, rand
import numpy as np
#from scipy import stats
from scipy import stats
import seaborn as sns
import matplotlib.pyplot as plt



# The function IBP_generator generates a graph Z ~ IBP(alpha)
#
# Input arguments :
# N : number of entities
# alpha : IBP parameter
#
# Output : Z
#
# Extra arguments:
# s=None: set s to the desired seed for numpy.random
# saveOpt=False: set to True for saving Z into the file filename
# displayOpt=True: set to True for comments
# filename='myIBP_Z.csv'
#
# Example: Z=IBP_generator(N=10,alpha=10.0,saveOpt=True,displayOpt=True,filename='myIBP_Z.csv')

def IBP_generator(N,alpha,s=None,saveOpt=False,displayOpt=True,filename='myIBP_Z.csv'):
    if s is None:
        pass
    else:
        seed(s)

    Z=[[] for i in range(N)]
    M=[]

    for i in range(N):
        # old dishes
        if i and len(Z[i-1]):
            for k in range(len(Z[i-1])):
                Z[i].append(rand()<=M[k]/float(i+1))
                M[k]+=Z[i][k]
        # new dishes
        n=poisson(alpha/float(i+1))
        Z[i]+=[1 for j in range(n)]
        M+=[1 for j in range(n)]

    K=len(Z[-1])
    for i in range(N-1):
        Z[i] += [0 for j in range(K-len(Z[i]))]

    if saveOpt:
        #for i in range(N):
        np.savetxt(filename,np.array(Z),fmt='%-u')
    if displayOpt:
        print ('Generated ',N,'x',K,'graph ~ IBP(',alpha,'):')
        print (np.array(Z))
        print ('Total number of entries :', sum(M), '( expectation :',alpha*N,')')
        if saveOpt:
            print ('Saved to ',filename)
        print

    return Z


# The function LinkProbabilityMatrix_generator generates a weight matrix W and a link probability matrix X for a graph Z generated by an IBP
#
#
# Input arguments:
# Z : graph generated by an IBP
# weight_sigma : standard deviation of the noise variables w_{kl}
#
# Output : [X,W]
#
# Extra arguments:
```

```
# s=None: set s to the desired seed for numpy.random
# saveOpt=False: set to True for saving X, W into the files filenameX, filenameW
# displayOpt=True: set to True for comments
# filenameX='myIBP_X.csv'
# filenameW='myIBP_W.csv'
#
# Example: [X,W]=LinkProbabilityMatrix_generator(IBP_generator(N=10,alpha=10.0,saveOpt=True,displayOpt=True,filename='myIBP_Z.csv'), 1.0,displayOp


def sigmoid(a):
    return 1 / float(1.0 + exp(-a))


def LinkProbabilityMatrix_generator(Z, weight_sigma, s=None, displayOpt=True, saveOpt=True, filenameX='myIBP_X.csv',
                                    filenameW='myIBP_W.csv'):
    if s is None:
        pass
    else:
        seed(s)

    N = len(Z)
    if not (N and len(Z[0])):
        print('The graph is not valid.')
    else:
        # Generation of W
        print("(generation of W)")
        K = len(Z[0])
        W = [None for k in range(K)]
        for k in range(K):
            W[k] = normal(0, weight_sigma, K)
        # Computation of X
        print("(generation of X)")
        X = np.dot(Z, W)
        X = np.dot(X, np.transpose(Z))
        for i in range(N):
            for j in range(N):
                X[i][j] = sigmoid(X[i][j])
        # X=[[None for j in range(N)] for i in range(N)]
        # ZWi=[None for l in range(K)]
        # for i in range(N):
        # for l in range(K):
        # ZWi[l]=0
        # for k in range(K):
        # ZWi[l]+=Z[i][k]*W[k][l]
        # for j in range(N):
        # X[i][j]=0
        # for l in range(K):
        # X[i][j]+=ZWi[l]*Z[j][l]
        # X[i][j]=sigmoid(X[i][j])

        if saveOpt:
            np.savetxt(filenameX, np.array(X), fmt='%.17f')
            np.savetxt(filenameW, np.array(W), fmt='%.17f')
        if displayOpt:
            print('Generated ', K, 'x', K, 'weight matrix W.')
            if saveOpt:
                print('Saved to ', filenameW)
            print()
            print('Generated ', N, 'x', N, 'link probability matrix X:')
            print(np.array(X))
            if saveOpt:
                print('Saved to ', filenameX)
        return [X, W]


# The function LinkMatrix_generator generates a link matrix Y for a given link probability matrix X
#
#
# Input arguments:
# X : link probability matrix
#
# Output : Y
#
# Extra arguments:
# s=None: set s to the desired seed for numpy.random
# saveOpt=False: set to True for saving X, W into the files filenameX, filenameW
# displayOpt=True: set to True for comments
# filename='myIBP_Y.csv'
#

def LinkMatrix_generator(X, s=None, displayOpt=True, saveOpt=True, filename='myIBP_Y.csv'):
    if s is None:
        pass
    else:
        seed(s)

    N = len(X)
    if not (N and len(X[0])):
        print('The graph is not valid.')
    else:
        # Generation of Y
```

```
    print("generation of Y")
    Y = rand(N, N)
    for i in range(N):
        for j in range(N):
            # x=rand()
            if (Y[i][j] <= X[i][j]):
                Y[i][j] = 1
            else:
                Y[i][j] = 0

    if saveOpt:
        np.savetxt(filename, np.array(Y), fmt='%-u')
    if displayOpt:
        print('Generated ', N, 'x', N, 'link matrix Y:')
        print(np.array(Y))
        if saveOpt:
            print('Saved to ', filename)
    return [Y]
```

Listing 16: Task 3.2 Q17 + Main + Graphs

```
if __name__ == "__main__":

    np.random.seed(10)
    N = 10
    alpha = 10.0
    Z_start = IBP_generator(N,alpha, saveOpt=True,displayOpt=True,filename = "myIBP_Z.csv")
    weight_sigma = 1
    [X_start,W_start] = LinkProbabilityMatrix_generator(Z_start,weight_sigma,displayOpt=True,saveOpt=True,filenameX=
                                                        'myIBP_X.csv',filenameW='myIBP_W')

    Y = LinkMatrix_generator(X_start, displayOpt=True, saveOpt=True, filename='myIBP_Y.csv')

    Z = IBP_generator(N,alpha)
    [X, W] = LinkProbabilityMatrix_generator(Z, weight_sigma, displayOpt=True, saveOpt=True)


    plt.subplot(121)
    with sns.axes_style("dark"):
        sns.set(font_scale=0.5)

        ax = sns.heatmap(Z_start, linewidth=.1, cmap=plt.cm.cubehelix_r, annot=True, linecolor="black", cbar=False)
        sns.set(font_scale=1)
        plt.title("True Z that generated data")
        plt.xlabel(" ")
        plt.ylabel(" ")
        bottom, top = ax.get_ylim()
        ax.set_ylim(bottom + 0.5, top - 0.5)
    plt.subplot(122)
    with sns.axes_style("dark"):
        sns.set(font_scale=1)
        ax = sns.heatmap(Y[0], linewidth=.1, cmap=plt.cm.cubehelix_r, annot=True, linecolor="black", cbar=False)
        plt.title("Y to infer")
        plt.xlabel(" ")
        plt.ylabel(" ")
        bottom, top = ax.get_ylim()
        ax.set_ylim(bottom + 0.5, top - 0.5)
    plt.savefig('generated_ibp.png')
    plt.show()

    n_iter= 20
    for i in range(n_iter):
        print("iteration",i)

        print(np.array(Z).shape)
        for j in range(len(Z)):
            Z,W = propose_Z(Z,W,Y,j)
            Z,W = propose_W(Z,W,Y,j)

    print("Z.shape : ",Z.shape)

    #print(y_given_rest(Z,W,Y))

    X= y_given_rest(Z,W,Y)
    Y_estimated = LinkMatrix_generator(X_start)


    print(np.array(Y_estimated[0])==np.array(Y[0]))
    print(np.sum(np.array(Y_estimated[0])==np.array(Y[0])))

    hits = np.sum(np.array(Y_estimated[0]==np.array(Y[0])))
    acc = hits /len(Y_estimated[0][0])/len(Y_estimated[0])
    print("accuracy of the Y_estimated : ", acc)

    plt.subplot(121)
    with sns.axes_style("dark"):
        sns.set(font_scale=0.5)
```

```
    ax = sns.heatmap(Z, linewidth=.1, cmap=plt.cm.cubehelix_r, annot=True, linecolor="black", cbar=False)
    sns.set(font_scale=1)
    plt.title("Result Z")
    plt.xlabel(" ")
    plt.ylabel(" ")
    bottom, top = ax.get_ylim()
    ax.set_ylim(bottom + 0.5, top - 0.5)
plt.subplot(122)
with sns.axes_style("dark"):
    sns.set(font_scale=1)
    ax = sns.heatmap(Y_estimated[0], linewidth=.1, cmap=plt.cm.cubehelix_r, annot=True, linecolor="black", cbar=False)
    plt.title("Y estimated")
    plt.xlabel(" ")
    plt.ylabel(" ")
    bottom, top = ax.get_ylim()
    ax.set_ylim(bottom + 0.5, top - 0.5)
plt.savefig('result_ibp.png')
plt.show()
#print(np.array(Z==Z_start))
```