Programming Assignment II : Metropolis Hastings Algorithm

Amatya Sharma
17CS30042

Note : A separate PDF of a sample run has been uploaded

Objective

Input

- n := number of iterations of algorithm
- K := number of bivariate gaussians for the mixture distribution
- $sigma$:= standard deviation for stepping normal distribution

Goal

- Build a GMM with the given values of means and covariances
- Draw n samples from the produced GMM using Metropolis-Hastings algorithm
- Plot them in $2D$

Output

For building dataset we build and plot:

- K - bivariate gaussians with means as $[2i, 3i]$ and covariances as $0.5 * sqrt(i) * I$ for all $1 <= i <= K$
- Uniform weighted mixture model of all K - bivariate gaussians

Metropolis-Hastings algorithm :

- Draw n samples from the produced GMM
- Plot the walk of sampled points along with rejected points
- Plot the accept points on $2D$ as well as $1D$ for both the dimensions for accepted points to visualize the sampled distribution

Dependencies and Notations in code part :

- `n` := number of iterations of algorithm
- `K` := number of bivariate gaussians for the mixture distribution
- `sigma` := standard deviation for stepping normal distribution
- `accepted` := points accepted by metropolis hastings algo
- `rejected` := pts rejected by metropolis algo
- `samples` := pts sampled by metroplois algo
- `rv[i]` := ith Gaussian distribution
- `mu[i]` := mean of `rv[i]`
- `cov[i]` := covariance of `rv[i]`
- `x1` := 1^{st} element of bivariate gaussian
- `x2` := 2^{nd} element of bivariate gaussian

Metropolis-Hasting Algorithm

Metropolis-Hastings algorithm is sampling algorithm to sample from high dimensional, which is otherwise difficult to sample directly (due to intractable integrals) distributions or functions.

It employs Markov Chain because to get the next sample, one only need to consider the current sample and Monte Carlo as it generates random sample which we could use to compute integrals or numerical results.

The core of the algorithm lies in the distribution $Q(x'|x)$, which is used to suggest the next candidate of the Markov Chain given the current state/sample, and the acceptance probability alpha which is used to decide whether we accept the new sample, or stay with the current sample.

The acceptance probability alpha is found by this equation:

$$alpha = min(1, \frac{P(x').Q(x|x')}{P(x).Q(x'|x)})$$

where :

- $P(x)$ is the target distribution = GMM
- Q is proposal distribution = Gaussian Distribution N(0, sigma)

In case of gaussian, transition distribution $Q(x|y)$ is symmetric i.e.

$$Q(x|x') = Q(x'|x)$$
$$\implies \frac{Q(x|x')}{Q(x'|x)} = 1$$
$$alpha = min(1, \frac{P(x').Q(x|x')}{P(x).Q(x'|x)})$$
$$\implies alpha = min(1, \frac{P(x')}{P(x)})$$

- We initialize the pts. by randomly sampling with a gaussian distribution with mean of first gaussian of the mizture and covariance s.t. it covers almost the entire distribution.

Implementation

We use function `def metropolis_hastings(p, n, sigma)` defined as follows:

```
def metropolis_hastings(p, n, sigma):
    x, y = np.random.multivariate_normal([2,3], [[math
.sqrt(1)/2,0],[0,math.sqrt(1)/2]]).T
    samples = np.zeros((n,2))  # has all the walk poin
ts both rej and acc
    accepted = []  # only new walk points
    rejected = []  # if no change has happended, predi
cteed pts xstar and ystar go to rej

    for i in range(n): # perform for n iterations
        x_star, y_star = np.array([x, y]) + np.random.
normal(scale=(sigma), size = 2)
        # sample a move such that d(x) ~ N(0,sigma)
        u=np.random.uniform(0,1)
        if u < p(x_star, y_star) / p(x, y): # accept a
nd move to new pt
            x, y = x_star, y_star
            accepted.append(np.array([x, y]))
        else :  # reject
            rejected.append(np.array([x_star, y_star
]))
        samples[i] = np.array([x, y])
    return np.array(accepted), np.array(rejected), np.
array(samples)
```

The functions works as follows:

- sample a move such that $d(x) \sim N(0, sigma)$
- check the proposal probability and consequently alpha
- randomly sample a value from $uniform\ [0, 1]$ distribution and make a move if alpha is more than the sample value

The function outputs :

- `samples[]` : has all the walk points both rej and acc
- `accepted[]` : only new walk points

- `rejected[]` : if no change has happended, predicteed pts xstar and ystar go to rej

Building Dataset

- We build K bivariate gaussians with given means and covariances using library function `scipy.stats.multivariable_normal()`
- Obtained GMM as a mean of all K pdfs
- Visualized both K-bivariate normal distributions and GMM

Running of Algorithm

- Build dataset as described
- Obtain probability distribution functions for GMM using function `def mixture_GM_pdf(x1, x2)`
- Obtain sampling using Metropolis-Hastings Algo as described using function `def metropolis_hastings(mixture_GM_pdf, n, sigma)`
- # Plot the data using `def visualize_data(rejected, samples, n, sigma, K)` function

```
def visualize_data(rejected, samples, n, sigma, K):
  print("Visualized Data on (n, sigma, K):", n, sig
ma, K)
  plt.plot(rejected[:,0], rejected[:,1], 'rx', labe
l = "Rejected", color = "red")
  plt.plot(samples[:,0], samples[:,1], label = "Wal
k")
  plt.title("Walk of Metropolis Algorithm")
  plt.xlabel("x1")
  plt.ylabel("x2")
  plt.legend()
  plt.grid()
  plt.show()

  print("\n First Direction Plot against no. of ite
rs:\n")
  iters=[i for i in range(1,n+1)]
  #print(yt)
  plt.plot(iters,samples[:,0])
  plt.xlabel('Iterations')
  plt.ylabel('%xdel1')
  plt.show()

  print("\n First Direction Frequency Plot against
 no. of iters:\n")
  plt.hist(samples[:,0],100)
  plt.xlabel('x1')
  plt.ylabel('Frequency of samples')
  plt.show()

  print("Sampled Points:\n")
  sns.jointplot(samples[:, 0], samples[:, 1])
  return
```

- Plot the accept points on $2D$
- Plot $1D$ for both the dimensions for accepted points to visualize the sampled distribution
- Plot $1D$ frequency histogram for first dimension
- Plot first dimension sampled pts against iterations of algorithm

```
def run_algo(n, K, sigma):
  # prepare position coordinates
  # these are just used to visualize data
  # no use in forming distibutions
  x, y = np.mgrid[0:3*K:.1, 0:4*K:.1]
  pos = np.dstack((x,y))

  # create k bivariate gaussians with required means a
nd covariances
  rv = {}
  mx = np.zeros(np.shape(x))
  i = 1
  mu =[]
  cov =[]
  while i<=K:
      mui = [2*i, 3*i]
      sigmai = np.eye(2)*0.5*np.sqrt(i)
      rv[i] = multivariate_normal(mui, sigmai)
      mx = np.add( mx, rv[i].pdf(pos) )
      plt.contour(x,y,rv[i].pdf(pos))
      mu.append(mui)
      cov.append(sigmai)
      i+=1

  def mixture_GM_pdf(x1, x2):
    i = 1
    mx = 0.0
    while i<=K:
      mx += rv[i].pdf([x1,x2])
      i += 1
    mx = mx/K
    return mx
```

```
    plt.title("Uniform Mixture Gaussian")
    plt.contour(x,y,mx)
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.show()

    accepted, rejected, samples = metropolis_hastings(mi
xture_GM_pdf, n, sigma)

    visualize_data(rejected, samples, n, sigma, K)

    return
```

Conclusions

- From the runs of algorithm we observe:
  - Usually the algorithm performs poor for very small values of K
  - For median values of K (~$5-10$) the samples closely resembled the original distribution
  - For very small values of $sigma$ $(\sim .01 - 1.5)$ the algorithm rejected a lot of samples and gave a poor outcome for median values of K $(\sim 10 - 15)$
  - Best approximations were observed at $sigma$ ~ $2.0 - 5.0$
  - Clearly increasing n is imporving results
- Metropolis-Hastings has an advantage over other methods of sampling such as Gibbs Sampling as we don't have to derive the conditional distributions analytically. We just need to know the joint distribution, and no need to derive anything analytically.
- Gibbs Sampling is much faster as compared to Metropolis-Hastings as s0ampling from conditional distribution is really fast, whereas sampling from full joint distribution is slow.
- While sampling the threshold value from uniform distribution, we can ignore the `min(1, x)` term for the alpha calculation, and just calculate `P(x') / P(x)`, because we only care about whether or not the ratio is bigger than some uniform random number `[0, 1]`, so when `P(x') / P(x) > 1`, it will then always satisfy the test just as `P(x') / P(x) = 1`, calculated from the `min(1, x)` term.
- Initializing with gaussian distribution s.t. the mean lies around the center of the distribution we are approximating helps sampling better.

References

[1] Peter Driscoll, “A comparison of least-squares and Bayesian fitting techniques to radial velocity data sets”

[2] Carson Chow, “MCMC and fitting models to data”

[3] John H. Williamson, “Data Fundamentals - Probabilities”

[4] Simon Rogers, “A first course in machine learning”

[5] Agustinus Kristiadi's Blog on Metropolis-Hastings

[6] Kevin P. Murphy. Machine Learning, A Probabilistic Perspective

[7] Christopher M. Bishop. Pattern Recognition and Machine Learning