Kernel Density Estimation using Gaussian Kernels

By: Amin Assareh

Background

In statistics, **maximum likelihood estimation** (MLE) is a method of estimating the parameters of a statistical model given observations, by finding the parameter values that maximize the likelihood of making the observations P(x) given the parameters θ. Prior to applying MLE, we need to make an assumption about the data distribution. A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters.

This concept is similar to using **Kernel Density Estimation** (KDE) which is intended to estimate the probability density function of a random variable using sampling distribution. However, KDE is a non-parametric approach in a sense that we don't make an assumption about the data distribution and any kernel function can be used for smoothing as long as it always produces a value greater than or equal to zero and it should integrate to 1 over the sample space.

Gaussian function can be used for both kernels and P(x) which makes MLE and KDE very similar in practice when applied with Gaussian Mixture Models.

Problem Statement

Here the task is to estimate the parameters of Gaussian distribution for MNIST and CIFAR datasets.

A **mixture model** decomposes the likelihood of data with respect to conditional probabilities on a latent variable z:

$$\log p(x) = \log \sum_{i=1}^{k} p(z_i) p(x|z_i)$$

(1)

$$p(x|z_i) = \prod_{j=1}^{d} p(x_j|z_i)$$

(2)

Gaussian mixture models further assume the probability of each data dimension follows a gaussian form:

$$p(x_j|z_i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp(-\frac{(x_j - \mu_{i,j})^2}{2\sigma_i^2})$$

(3)

For simplicity lets assumes probability of each of the k latent variables are the same:

$$p(z_i) = \frac{1}{k}$$

(4)

After plugging in the above formulas, we would have:

$$\log p(x) = \log \sum_{i=1}^{k} \exp\{\log \frac{1}{k} + \sum_{i=1}^{d} [-\frac{(x_j - \mu_{i,j})^2}{2\sigma^2} - \frac{1}{2}\log(2\pi\sigma^2)]\}$$

(5)

So for each example in the validation set, we get k examples in the train set and use their values for μ in the above formula. Finally, the log probability of the total validation set of size m can be written as:

$$\mathcal{L}_{\mathcal{D}_B} = \frac{1}{m} \log \prod_{i=1}^{m} p(x_i^B) = \frac{1}{m} \sum_{i=1}^{m} \log p(x_i^B)$$

(6)

Here we want to implement this function for MNIST and CIFAR dataset.

Solution

The code kernel_density_estimate.py inputs one of those datasets and number of samples to use to find the optimal σ and outputs:

- likelihood estimates over different potential values of a sigma (using a subset of data)
- the sigma that results in the optimal likelihood
- Actual likelihood estimate and time taken using the best sigma (using the full dataset)

Here is the syntax to run the code ("cifar" can be used instead of "mnist"):

"python kernel_density_estimate.py --data "mnist" --sample_size 10000"

Let's go over the main component of the code which is kde_compute function and used to implement the likelihood formula 5 and 6:

Looking at the above formula (5) we can see there are 3 terms that we need to compute:

$$\log p(x) = \log \sum_{i=1}^{k} \exp\{\log \frac{1}{k} + \sum_{i=1}^{d} [-\frac{(x_j - \mu_{i,j})^2}{2\sigma^2} - \frac{1}{2}\log(2\pi\sigma^2)]\}$$

Finally we will have one more sum one over validation set (m) per equation (6).

The first term in equation (5) is constant for all different values of σ so we can omit that in our optimization, however I just keep it to have a better estimate for likelihood. Let's call the 3 terms terms_0, term_1 and term_2 in the same order they appear in the formula.

The kde_compute() function below implements the above formula:

```
def kde_compute(sigma, D_train, D_val):

    likelihood =  0

    train_size, val_size, num_features = len(D_train), len(D_val), len(D_train[0])

    term_0 = -Decimal(train_size).ln()

    term_2 = -Decimal(0.5 * num_features) * Decimal(2 * pi * (sigma ** 2)).ln()

       #loop over validation examples (m in equation 6)

    for i in  range(0, val_size):

       #sum over features (d in equation 5)

        w = np.sum((-((np.matlib.repmat(D_val[i], train_size, 1).astype(np.float64) -
D_train.astype(np.float64)) ** 2)) / (2 * (sigma ** 2)), axis=1)

        wmax = np.max(w)
```

```
        #sum over train examples (k in equation 5)
         term_1 = Decimal(wmax + np.log(np.sum(np.exp(v-m))))
         likelihood +=  term_0 + term_1 + term_2
    return likelihood / val_size
```

Since numpy.exp(x) can handle limited precision and for large negative values for x it returns 0, I mathematically transform the log of sums over exponentials the following way to avoid exp() of large numbers:

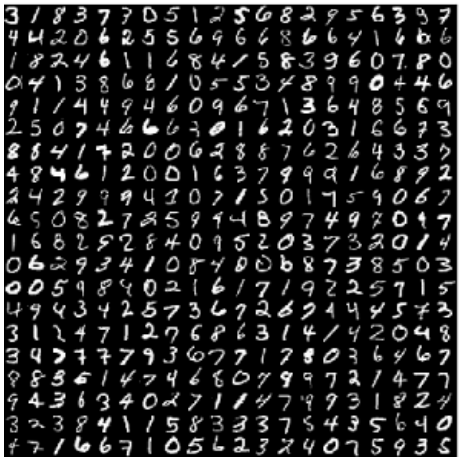
log(sum(exp(w)))= log(sum(exp(w-wmax)*exp(wmax)))= wmax + log(sum(exp(w-wmax)))

Other components of the code are as follows:

load_data_mnist(): loads the MNIST data, assigns 10k examples for train, 10k for validation and the rest for test

load_data_cifar(): loads CIFAR data, assigns 10k for train, 10k for validation and the same test set

Viz(): reshapes the data to (R*n, C*n, D) dimension, in which R is the number of images to show in rows, C is the number of images to show in columns, n is the number of pixels in either x or y of square images and D is the number of channels. The function then uses pyplot.imshow() function to show the images.



Results:

Here are the results of running the code for 2 datasets to find the best sigma using a sample size of 1000 examples:

Sigma	CIFAR Validation Likelihood	MNIST Validation Likelihood
0.05	-17635.27	- 4916.183
0.08	-4444.861	-1300.88
0.1	-1753.222	-556.492
0.2	**620.2744**	**125.8678**
0.5	-933.745	-249.2087
1	- 2882.84	-743.2871
1.5	-4094.843	-1051.48
2	-4966.536	-1272.944

Here is the total time taken for each dataset when fitting KDE using the optimal sigma:

	Likelihood	Time Taken
CIFAR	857.4606	8085.2066 sec
MNIST	236.1038	2808.7035 sec

Conclusions

Kernel Density Estimation has a parameter called bandwidth which determines the degree of smoothing (number of points around each location which are used to estimate the density function at that location). When bandwidth too small it results in non-smooth and noisy looking pdf curves and when it's too large it cannot show the informative patterns in the actual distribution. Applying maximum likelihood estimation of mixture of gaussian to optimize the standard deviation is similar to optimizing the bandwidth parameter in KDE.

Here we applied the same concept on two well-known image datasets of CIFAR and MNIST. As we can see in the results, σ =0.2 gives the best results and corresponds to the best window size in both datasets. CIFAR has an extra dimension that encodes the color and makes the processing time longer than MNIST. The code was structured by 3 main steps of 1) loading the data, 2) visualization and 3) density estimation. Since step 3 takes up most of the running time, here we tried to go over the implementation details of it. We used vectorization as much as possible to optimize the running time and used for loops only once.