Statistical Machine Learning: Exercise 2

Group A: Kexin Wang (2540047), Paul Philipp Seitz (2337506)

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer Term 2022

Task 1: Optimization (25 Points)

1a) Numerical Optimization (15 Points)

Derivative of the function:

$$\frac{\partial f}{\partial x_j} = \sum_{i=1}^{N} 200(x_i - x_{i-1}^2)(\epsilon_{i,j} - 2x_{i-1}\epsilon_{i-1,j}) - 2(1 - x_{i-1})\epsilon_{i-1,j}$$

$$= 200(x_j - x_{j-1}^2) - 400x_j(x_{j+1} - x_j^2) - 2(1 - x_j)$$

but the first and the last x are exception:

$$\frac{\partial f}{\partial x_0} = -400x_0(x_1 - x_0^2) - 2(1 - x_0)$$

$$\frac{\partial f}{\partial x_{N-1}} = 200(x_{N-1} - x_{N-2}^2)$$

```python
import numpy as np
import matplotlib.pyplot as plt

def rosenbrock(x):
    sum = 0
    for i in range(len(x)-1):
        sum += 100*(x[i+1] - x[i]**2)**2 + (x[i] - 1)**2

    return sum

def rosenbrock_gradient(x, n):
    grad = np.zeros(n)

    for i in range(n-1):
        grad[i] = 2*(x[i]-1) - 400*x[i]*(x[i+1]-x[i]**2)

        if i != 0:
            grad[i] += 200*(x[i]-x[i-1]**2)

    grad[n-1] = 200*(x[n-1]-x[n-2]**2)
    return grad


def loss(x):
    return np.linalg.norm(rosenbrock(x))

```

```python
def gradient_descent(x0, alpha, max_iter):
    x = x0
    history = [loss(x)]

    for i in range(max_iter):
        x = x - alpha*rosenbrock_gradient(x, len(x))
        current_loss = loss(x)
        history.append(current_loss)

    return x, history

def optimize(alpha):
    np.random.seed(7431)
    x0 = np.random.rand(20)
    max_iter = 10000
    x, history = gradient_descent(x0, alpha, max_iter)

    # Plot history
    plt.figure(figsize=[8,6])
    plt.plot(history, linewidth=3.0)
    plt.legend(['Loss with alpha=%.3f' % alpha],fontsize=18)
    plt.title('GD Learning Curve',fontsize=16)
    plt.show()

# Higher learning rate leads to divergence
# Lower learning rate leads to slower progress and therefor higher loss after 10000 steps
optimize(1e-3)
```

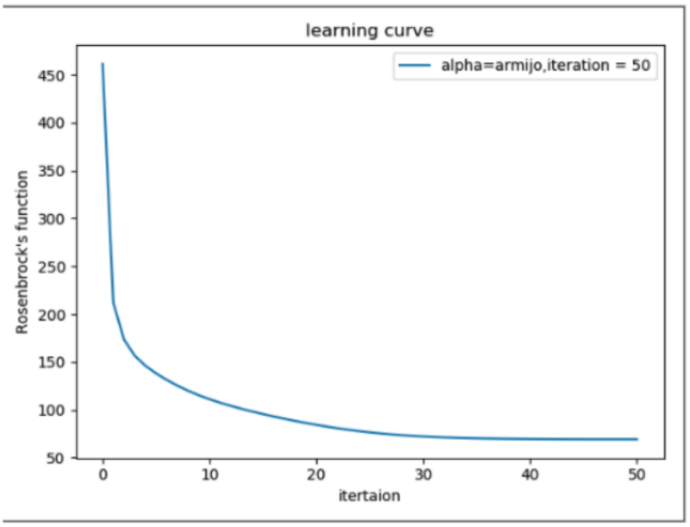

Figure 1: gradient descent algorithm after 10,000 steps on Rosenbrock's function

1b) Gradient Descent Variants (10 Points)

1).A. Batch Gradient Descent, BGD[3] :

They process all samples simultaneously in a large batch, i.e. use all samples for gradient update at each iteration.

Advantages:

- During training, a fixed learning rate is used and there is no need to worry about the appearance of learning rate recession.
- The direction determined by the full data set is more representative of the total sample and thus more accurate towards the direction where the extremes are located. When the objective function is convex, it must converge to the global minimum, and if the objective function is non-convex it converges to the local minimum.
- It has an unbiased estimate of the gradient. The more samples there are, the lower the standard deviation.
- One iteration is computed for all samples, and at this time, parallelism is achieved by using vectorization for the operation.

Disadvantages:

- Although vectorization is used in the computation process, it still takes a lot of time to traverse all the samples, especially when the data set is large (millions or even hundreds of millions), it is a bit difficult to complete.
- Since each update occurs after traversing the entire sample, it is only after all of them are completed that some examples may be redundant and not very useful for parameter updating.In terms of the number of iterations, the number of BGD iterations is relatively small.

In terms of the number of iterations, the number of BGD iterations is relatively small.

B. Stochastic Gradient Descent, SGD[1] :

Stochastic Gradient Descent uses a sample for parameter update at each iteration (mini-batch size = 1).

Advantages:

- Noise is added to the learning process, which improves the generalization error.
- Since the loss function is not on all training data, but on a randomly optimized loss function on one training data in each iteration, the parameters are updated much faster in each round.

Disadvantages

- Does not converge and fluctuates around the minimum value.
- Cannot use vectorization calculations in one sample, and the learning process becomes slow.
- A single sample does not represent the trend of the whole sample.
- When local minima or saddle points are encountered, SGD gets stuck at a gradient of 0.

C. Mini-batch Gradient Descent, MBGD:

MBGD is in between the above two, using more than one but not all training samples.In each step of the algorithm, we randomly draw a small batch of samples from a training set of m samples (which have been disordered). Usually uses a batch of 32 or 64 samples.

Advantages

- The computation is faster than Batch Gradient Descent because the update can be performed by traversing only some samples.
- Random sample selection helps to avoid repeating redundant samples and samples that contribute less to parameter updates.
- Using one batch at a time can greatly reduce the number of iterations required for convergence, and at the same time can make the converged result closer to the effect of gradient descent.

Disadvantages

- During the iteration, the learning process fluctuates because of the presence of noise. Therefore, it hovers in the region of the minimum and does not converge.
- The learning process will have more oscillations. To get closer to the minimum, the learning rate decay needs to be increased to reduce the learning rate and avoid excessive oscillations.

- choice of batch size may cause some problems.

If the sample size is relatively small, the batch gradient descent algorithm is used. If the sample is too large, or online algorithm, use stochastic gradient descent algorithm. In the practical general case, use the small batch gradient descent algorithm.

2).Momentum[2] :

Momentum methods are designed to accelerate learning (speed up gradient descent), especially when dealing with high curvature, small but consistent gradients, or gradients with noise. The Momentum algorithm accumulates a moving average of the previous exponential decay of the gradient and continues to move in that direction.

The idea of Momentum is simply to update a portion more of the updates from the previous iteration to smooth out the gradient for this iteration. From a physical point of view, it is like a small ball rolling down is affected by its own historical momentum, that is why it is called Momentum algorithm.

This has the direct effect of making the turnaround less dramatic when the gradient is falling, so that it can rush more smoothly and quickly to the local minimum. For example, in the SGD algorithm with momentum, when the general value of the momentum parameter is changed, it may be 2, 10, or 100 times faster than the SGD algorithm. And it does improve the convergence rate of some algorithms, for example, in the case of convex batch gradients, Nesterov momentum improves the error convergence rate. But not all algorithms are applicable,which means momentum is not always useful. for insance, in the case of stochastic gradient, Nesterov momentum does not improve the convergence rate.

Task 2: Density Estimation (35 Points)

2a) Gaussian Maximization Likelihood Estimate (10 Points)

$$L(x_1,...,x_n|\mu,\Sigma)=\prod_{i=1}^{n}p(x_i|\mu,\Sigma)=\prod_{i=1}^{n}\frac{1}{(2\pi)^{D/2}}\frac{1}{|\Sigma|^{1/2}}e^{-\frac{1}{2}(x_i-\mu)^T\Sigma^{-1}(x_i-\mu)}$$

Using Log-Likelihood to determine ML estimates for the mean and covariance of the multivariate Gaussian distribution.

$$\widetilde{L}(x_1,...,x_n|\mu,\Sigma)=\ln L(x_1,...,x_n|\mu,\Sigma)=\sum_{i=1}^{n}\ln p(x_i|\mu,\Sigma)$$

$$\ln p(x_i|\mu,\Sigma)=-\frac{1}{2}(D\ln(2\pi)+\ln(|\Sigma|)+(x_i-\mu)^T\Sigma^{-1}(x_i-\mu))$$

$$\widetilde{L}(x_1,...,x_n|\mu,\Sigma)=-\frac{n}{2}(D\ln(2\pi)+\ln(|\Sigma|))-\frac{1}{2}\sum_{i=1}^{n}(x_i-\mu)^T\Sigma^{-1}(x_i-\mu))$$

We begin with μ_{MLE}:

$$\frac{\partial\widetilde{L}}{\partial\mu}=-\frac{1}{2}\sum_{i=1}^{n}\frac{\partial}{\partial\mu}[(x_i-\mu)^T\Sigma^{-1}(x_i-\mu))]\overset{(86)}{=}\sum_{i=1}^{n}\Sigma^{-1}(x_i-\mu)=0$$

$$\sum_{i=1}^{n}\Sigma^{-1}x_i=n\Sigma^{-1}\mu$$

$$\mu_{MLE}=\frac{1}{n}\sum_{i=1}^{n}x_i$$

Now for Σ_{MLE}:

$$\frac{\partial\widetilde{L}}{\partial\Sigma}=-\frac{n}{2}\frac{\partial}{\partial\Sigma}\ln(|\Sigma|)-\frac{1}{2}\sum_{i=1}^{n}\frac{\partial}{\partial\Sigma}[(x_i-\mu)^T\Sigma^{-1}(x_i-\mu))]$$

$$-\frac{n}{2}\frac{\partial}{\partial\Sigma}\ln(|\Sigma|) = -\frac{n}{2}\frac{1}{|\Sigma|}\frac{\partial}{\partial\Sigma}|\Sigma| \overset{(49)}{=} -\frac{n}{2}\frac{1}{|\Sigma|}|\Sigma|(\Sigma^{-1})^T \overset{(\Sigma^{-1}symmetric)}{=} -\frac{n}{2}\Sigma^{-1}$$

$$\frac{\partial}{\partial\Sigma}[(x_i-\mu)^T\Sigma^{-1}(x_i-\mu))] \overset{(61)}{=} -\Sigma^{-T}(x_i-\mu)(x_i-\mu)^T\Sigma^{-T} \overset{(\Sigma^{-1}symmetric)}{=} -(x_i-\mu)(x_i-\mu)^T\Sigma^{-1}\Sigma^{-1}$$

$$\frac{\partial\widetilde{L}}{\partial\Sigma} = -\frac{n}{2}\Sigma^{-1} + \frac{1}{2}(\sum_{i=1}^{n}(x_i-\mu)(x_i-\mu)^T)\Sigma^{-1}\Sigma^{-1} = 0$$

$$(\sum_{i=1}^{n}(x_i-\mu)(x_i-\mu)^T)\Sigma^{-1}\Sigma^{-1} = n\Sigma^{-1}$$

$$\Sigma_{MLE} = \frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)(x_i-\mu)^T$$

2b) Prior Probabilities (2 Points)

The prior probability $p(A) = \frac{count(A)}{count(A)+count(B)}$ of each class A from the dataset is computed in the python code below.

```
def get_priors(d1,d2):
   #counting the number of class data point than normalizing
   #d1,d2 are the data load from txt files.
   p1 = 1. * d1.shape[0] / (d1.shape[0] +d2.shape[0])
   p2 = 1. * d2.shape[0] / (d1.shape[0] +d2.shape[0])
   return p1,p2
```

The results are shown as the following figure. The prior possibility of dataset 1 is the $p(C_1) = 0.24$ and the prior possibility of dataset 2 is $p(C_2) = 0.76$

```
0.23847695390781562
0.7615230460921844
```

Figure 2

2c) Biased ML Estimate (5 Points)

The bias of an estimator is the difference between this estimator's expected value and the true value of the parameter being estimated. We construct some estimator $\hat{\theta}$ that maps observed data to values that we hope are close to θ. The bias of $\hat{\theta}$ relative to θ is defined as

$$Bias(\hat{\theta},\theta) = Bias_\theta[\hat{\theta}] = E_{x|\theta}[\hat{\theta}] - \theta = E_{x|\theta}[\hat{\theta}-\theta].$$

Where $E_{x|\theta}$ denotes expected value over the distribution $P(x|\theta)$

```
def estimate_parameters(data):
   covar = np.zeros((data.shape[1], data.shape[1]))

   N = data.shape[0]
   mu = np.sum(data, axis=0) / N
   for xi in data:
      diff = xi - mu
      covar += np.outer(diff, diff.T)

   covar_unbiased = covar / (N - 1)
   covar /= N

   return mu, covar, covar_unbiased
```

```
mu1, covar1, covar_unb1 = estimate_parameters(data_1.values)
mu2, covar2, covar_unb2 = estimate_parameters(data_2.values)
print("mean1=" ,mu1)
print("biasedcovariance1=",covar1)
print("unbiasedcovariance1=",covar_unb1)
print("mean2=",mu2)
print("biasedcovariance2=",covar2)
print("unbiasedcovariance2=",covar_unb2)
```

After computing them, the results are given as the following figure.

```
mean1= [-0.70623122 -0.81921721]
biasedcovariance1= [[9.05734193 2.68490698]
 [2.68490698 3.60344808]]
unbiasedcovariance1= [[9.09555856 2.6962357 ]
 [2.6962357  3.61865251]]
mean2= [3.98460051 3.98500772]
biasedcovariance2= [[4.18045643 0.02797194]
 [0.02797194 2.75628915]]
unbiasedcovariance2= [[4.18596428 0.02800879]
 [0.02800879 2.75992063]]
```

Figure 3

2d) Class Density (5 Points)

```
def multivariate_gaussian(data, mu, covar):
    """
    return likelihood for all given samples
    """
    out = np.empty(data.shape[0])
    # denominator
    y = np.sqrt((2 * math.pi) * np.linalg.det(covar))

    # compute for each datapoint
    for i, x in enumerate(data):
        diff = x - mu
        out[i] = np.exp(-0.5 * diff.T.dot(np.linalg.inv(covar)).dot(diff)) / y

    return out

# Visualization of class density
def visualize_class_density(mu, covar, data):
    steps = 100

    x_data = data[:, 0]
    y_data = data[:, 1]

    x_min = x_data.min()
    x_max = x_data.max()
    y_min = y_data.min()
    y_max = y_data.max()

    x = np.arange(x_min - 1, x_max + 1, (x_max - x_min + 2) / steps)
    y = np.arange(y_min - 1, y_max + 1, (y_max - y_min + 2) / steps)

    Y, X = np.meshgrid(y, x)
```

```
    Z = np.empty((steps, steps))

    for i in range(n_models):
        for j in range(steps):
            # construct vector with same x and all possible y to cover the plot space
            points = np.append(X[j], Y[j]).reshape(2, x.shape[0]).T
            Z[j] = multivariate_gaussian(points, mu[i], covar[i])
        c_plot = plt.contour(X, Y, Z)
        plt.clabel(c_plot, inline=1, fontsize=10)

mus = np.append(mu1, mu2).reshape(n_models, mu1.shape[0])
sigmas = np.append(covar_unb1, covar_unb2).reshape(n_models, covar_unb1.shape[0],
    covar_unb1.shape[1])
data = np.append(data_1, data_2, axis=0)
priors = np.append(prior1, prior2).reshape(n_models, 1)

plt.figure(0)
visualize_class_density(mus, sigmas, data)

# plot the samples
plt.plot(data_1.values[:, 0], data_1.values[:, 1], 'co', zorder=1, color="pink",
    label="Density1")
plt.plot(data_2.values[:, 0], data_2.values[:, 1], 'cx', zorder=1, label="Density2")
plt.xlabel('x')
plt.ylabel('y')
plt.title("2d) Class Density: Estimate of Unbiased Gaussian Distributions")
plt.legend()
```

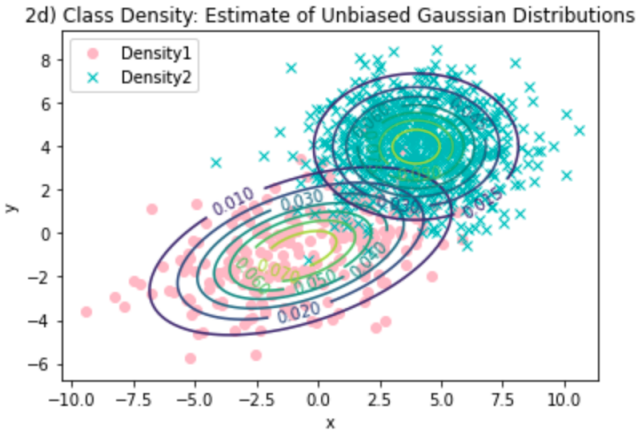

Figure 4

2e) Posterior (8 Points)

```
def visualize_posterior(mu, covar, data, prior):
    steps = 100

    # x_data = data[:, 0]
    # y_data = data[:, 1]

    x_data = np.arange(-15,15,30/998)
    y_data = data[:, 1]

```

```
    x_min = x_data.min()
    x_max = x_data.max()
    y_min = y_data.min()
    y_max = y_data.max()

    x = np.arange(x_min - 1, x_max + 1, (x_max - x_min + 2) / steps)
    y = np.arange(y_min - 1, y_max + 1, (y_max - y_min + 2) / steps)

    Y, X = np.meshgrid(y, x)
    Z = np.empty((steps, steps))

    for i in range(n_models):
        for j in range(steps):
            # construct vector with same x and all possible y to cover the plot space
            points = np.append(X[j], Y[j]).reshape(2, x.shape[0]).T
            Z[j] = multivariate_gaussian(points, mu[i], covar[i]) * prior[i]
        c_plot = plt.contour(X, Y, Z)
        plt.clabel(c_plot, inline=1, fontsize=10)

plt.figure(1)
    visualize_posterior(mus, sigmas, data, priors)
plt.plot(data_1.values[:, 0], data_1.values[:, 1], 'co', zorder=1, color="grey",
    label="Density1")
plt.plot(data_2.values[:, 0], data_2.values[:, 1], 'cx', zorder=1, color="yellow",
    label="Density2")
plt.xlabel('x')
plt.ylabel('y')
plt.title("2e) Posterior: Posterior Distributions and Decision Boundary")
plt.legend()
plt.show()
```

The below plot shows the posterior distribution of each class $P(C_i|x)$ and the decision boundary.

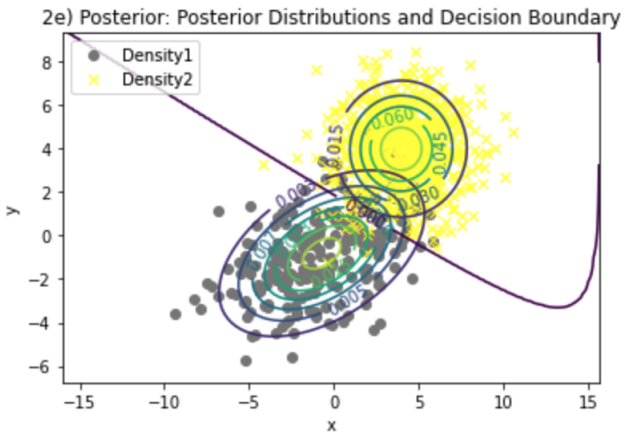

Figure 5

As shown in the figure, the decision boundary is not linear because the covariance of Gaussian is not identical.

Task 3: Non-parametric Density Estimation (20 Points)

3a) Histogram (4 Points)

```
import numpy as np
from matplotlib import pyplot as plt

train_data= np.loadtxt('nonParamTrain.txt')
test_data= np.loadtxt('nonParamTest.txt')

binwidth = 0.02
n,bins,patches = plt.hist(train_data,bins=np.arange(min(train_data),max(train_data),binwidth))
```

for binwidth=0.02, the figure is shown as the following figure.

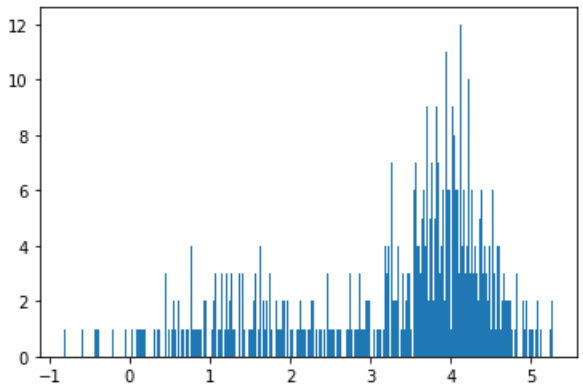

Figure 6

```
binwidth = 0.5
n,bins,patches = plt.hist(train_data,bins=np.arange(min(train_data),max(train_data),binwidth))
```

for binwidth=0.5, the figure is shown as the following figure.

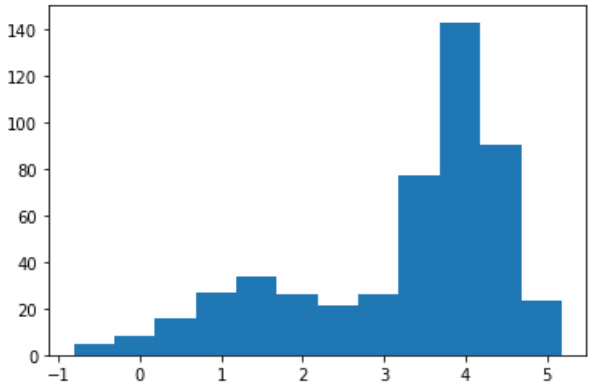

Figure 7

```
binwidth = 2.0
n,bins,patches = plt.hist(train_data,bins=np.arange(min(train_data),max(train_data),binwidth))
```

for binwidth=2.0, the figure is shown as the following figure.

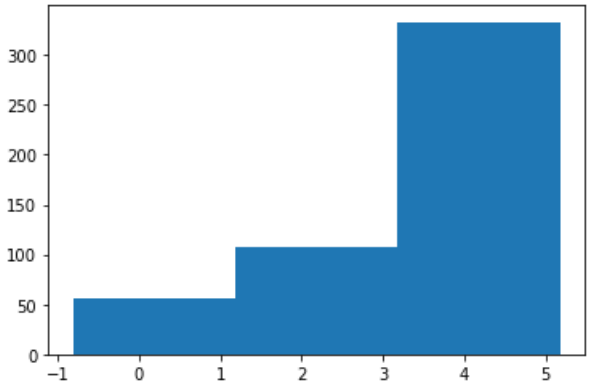

Figure 8

The bin size of 0.5 performs best. The 0.02 seems to be overfitting the data as it is not smooth enough. In contrast, 2.0 might be too smooth and could thus capture too little data and be too vague. However, we cannot easily determine which bin size results in the highest quality of estimation without knowing the underlying model.

3b) Kernel Density Estimate (6 Points)

```
import numpy as np
import matplotlib.pyplot as plt

train_data = np.loadtxt('nonParamTrain.txt')
test_data = np.loadtxt('nonParamTest.txt')

x = np.linspace(-4,8,1000)

def kde(x,h,data):
   distance = (np.repeat([data],np.size(x),0).T - np.repeat([x],np.size(data),0))**2
   p = np.sum(np.exp(-distance/2/h**2),0)
   p = p*(1/np.sqrt(2*np.pi)/h/np.size(data))
   return p

p1x = kde(x,0.03,train_data)
p2x = kde(x,0.2,train_data)
p3x = kde(x,0.8,train_data)

fig, ax = plt.subplots()
ax.plot(x,p1x)
ax.plot(x,p2x)
ax.plot(x,p3x)
plt.legend(['h = 0.03','h =0.2','h=0.8'])
plt.xlabel('x')
plt.ylabel('p(x)')
```

From the above figures, it can be seen that $\sigma = 0.2$ performs best since it has low fluctuations with an acceptable log-likelihood.
When $\sigma = 0.8$ the denisty estimation is too much smoothed and does not represent the characteristics of the original distribution.
For $\sigma = 0.03$ even though the log-likelihood is most desirable, the estimate fluctuates strongly which results in the graph being too noisy (overfitting).

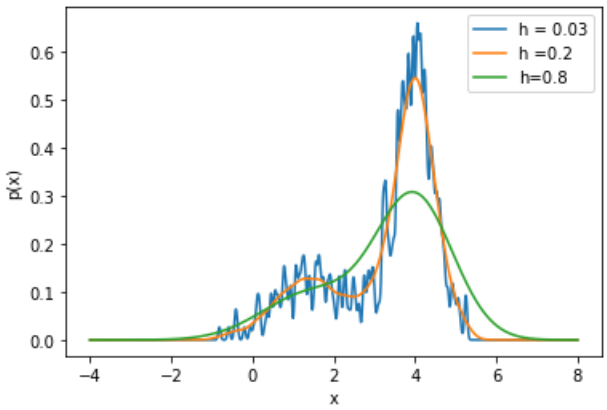

Figure 9: KDE

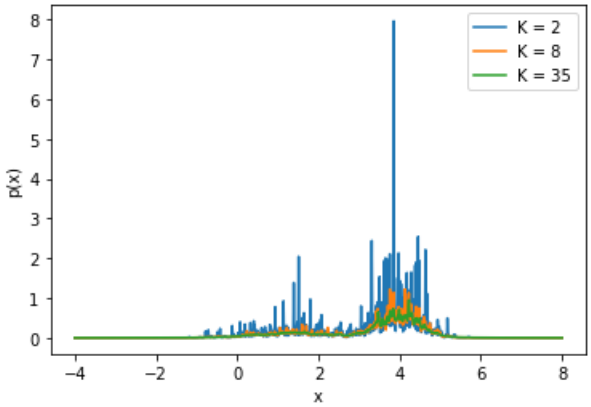

Figure 10: KDE_Test

3c) K-Nearest Neighbors (6 Points)

```
import numpy as np
import matplotlib.pyplot as plt

train_data = np.loadtxt('nonParamTrain.txt')
test_data = np.loadtxt('nonParamTest.txt')

x = np.linspace(-4,8,1000)

def knn(x,K,data):
   distance = abs(np.repeat([data],np.size(x),0).T - np.repeat([x],np.size(data),0))
   distance.sort(axis=0)
   V = []
   p = []
   for i in range(np.size(x)):
      V.append(2*distance[K-1,i])
      p.append(K/(np.size(data)*V[i]))
   return np.array(p)

p1x = knn(x,2,train_data)
p2x = knn(x,8,train_data)
p3x = knn(x,35,train_data)

```

```
fig, ax = plt.subplots()
ax.plot(x,p1x)
ax.plot(x,p2x)
ax.plot(x,p3x)
plt.legend(['K = 2','K = 8','K = 35'])
plt.xlabel('x')
plt.ylabel('p(x)')
```

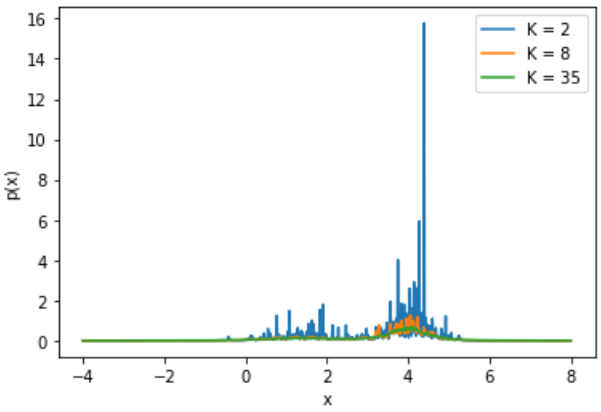

Figure 11: KNN

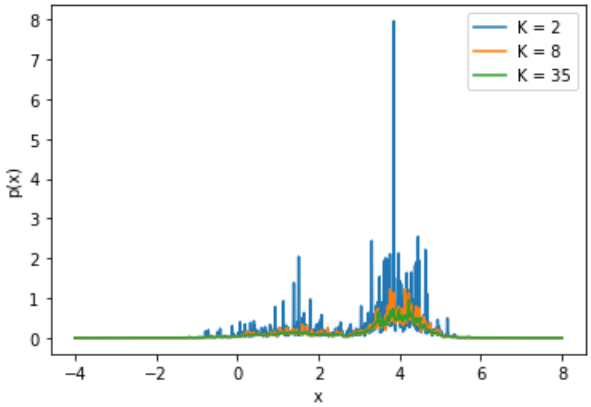

Figure 12: KNN_Test

When $K = 2$ we get noisy results and overfit, when $K = 8$ we still get a bit noisy results and overfit and when $K = 35$ the results are better.

3d) Comparison of the Non-Parametric Methods (4 Points)

The purpose of training machine learning is to be able to better predict new unknown samples, which means the generalization capability plays a very significant role. If the machine learning algorithm learns only for the training datasets, what it actually learns may be noise. And if the model obtained using the training datasets can get good results on the training datasets, but poor results in the face of the test datasets, then it means that the model generalization capability is very weak, for this case, we generally called **overfitting**.

What we are really looking for is the model with the best generalization capability. So measuring the model generalization capability is what makes it more meaningful to divide the datasets additional to the test datasets.
The log-likelihood of the testing data is estimated using the KDE estimators and the KNN estimator. We need to test

them ob a different data set to avoid the problem of overfitting the model.
Log-likelihoods of the estimators for both training and testing sets are giving in the table below:

KDE Estimators			KNN Estimators		
	Training set	Testing Set		Training set	Testing Set
h=0.03	2425.84	-inf	k=2	-1256.04	-1672.38
h=0.2	2383.64	-7853.36	k=8	-1127.73	-1544.07
h=0.8	2305.52	853.49	k=35	-949.19	-1356.53

Table 1

The test set is used to predict the performance of the model on previously unseen data. We would choose the KNN estimator with k=2 because it has the best log-likelihood on the testing set with least degree of overfitting.

Task 4: Expectation Maximization (20 Points)

4a) Gaussian Mixture Update Rules (2 Points)

The model parameters are the means, the covariance and the responsibility of each mixture component.

At $E-step$, we compute the posterior distribution, also called responsibility, for each mixture component and for all data point:

$$\alpha_n j = p(j|x_n) = \frac{\pi_j N(X_n|\mu_j, \sigma_j)}{\sum_{i=1}^{N} \pi_i N(X_n|\mu_i, \sigma_i)}$$

At $M-step$, we compute the model parameters:

$$\mu_j = \frac{\sum_{i=1}^{N} \alpha_{nj} x_n}{\sum_{i=1}^{N} \alpha_{nj}}, \sigma_j^2 = \frac{\sum_{i=1}^{N} \alpha_{nj}(x_n - m_j)(x_n - m_j)^{\mathrm{T}}}{\sum_{i=1}^{N} \alpha_{nj}}, \pi_j = \frac{\sum_{i=1}^{N} \alpha_{nj}}{N}.$$

4b) EM (18 Points)

The Expectation-Maximization algorithm for Gaussian Mixture Models is implemented and initialized uniformly. The multivariate normal probability density function is computed with the help of **multivariate_normal** function from *scipy.stats*. The code is given below performs this operation.

```
import math

import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

training_data = pd.read_csv("gmm.txt", sep=" ")
k = 4
n_iter = 30


def main():
    # init
    covar = np.array(
        [np.identity(training_data.shape[1]) for _ in range(k)])
    mu = np.random.uniform(training_data.min().min(), training_data.max().max(),
```

```
                    size=(k, training_data.shape[1]))
    pi = np.random.uniform(size=(k,))

    log_likelihood = np.empty((n_iter,))

    for i in range(n_iter):
        alpha = e(x=training_data.values, mu=mu, covar=covar, pi=pi)
        mu, covar, pi = m(x=training_data.values, alpha=alpha)

        # plot at given steps
        if i + 1 in [1, 3, 5, 10, 30]:
            plt.figure(i)
            visualize(mu, covar, training_data.values, i)
        log_likelihood[i] = calculate_log_likelihood(training_data.values, mu, covar, pi)

        print("Finished iteration {}".format(i))

    plt.figure(n_iter + 1)
    plt.plot(log_likelihood)
    plt.xlabel("Iterations")
    plt.ylabel("Log-Likelihood")
    plt.title("Log-Likelihood Progress")
    plt.show()


def e(x, mu, covar, pi):
    alpha = np.empty((k, x.shape[0]))
    for i in range(k):
        alpha[i] = pi[i] * multivariate_gaussian(x, mu[i], covar[i])

    # sum over all models per data point
    denominator = np.sum(alpha, axis=0)

    return alpha / denominator


def m(x, alpha):
    N = np.sum(alpha, axis=1) # sum over all data points per model

    mu = np.zeros((k, x.shape[1]))
    covar = np.zeros((k, x.shape[1], x.shape[1]))

    for i in range(k):
        # update mu
        for j, val in enumerate(x):
            mu[i] += (alpha[i, j] * val)

        mu[i] /= N[i]

        # update covariance
        for j, val in enumerate(x):
            diff = val - mu[i]
            covar[i] += alpha[i, j] * np.outer(diff, diff.T)

        covar[i] /= N[i]

    # update pi
    pi = N / x.shape[0]

    return mu, covar, pi
```

```
77
78
79  def multivariate_gaussian(data, mu, covar):
80
81      out = np.empty(data.shape[0])
82      denominator = np.sqrt((2 * math.pi) * np.linalg.det(covar))
83      covar_inv = np.linalg.inv(covar)
84
85      # compute for each datapoint
86      for i, x in enumerate(data):
87          diff = x - mu
88          out[i] = np.exp(-0.5 * diff.T.dot(covar_inv).dot(diff)) / denominator
89
90      return out
91
92
93  def visualize(mu, covar, data, iteration):
94
95      steps = 100
96
97      x_data = data[:, 0]
98      y_data = data[:, 1]
99
100     x_min = x_data.min()
101     x_max = x_data.max()
102     y_min = y_data.min()
103     y_max = y_data.max()
104
105     x = np.arange(x_min - 1, x_max + 1, (x_max - x_min + 2) / steps)
106     y = np.arange(y_min - 1, y_max + 1, (y_max - y_min + 2) / steps)
107
108     Y, X = np.meshgrid(y, x)
109     Z = np.empty((steps, steps))
110
111     for i in range(k):
112         for j in range(steps):
113             # construct vector with same x and all possible y to cover the plot space
114             points = np.append(X[j], Y[j]).reshape(2, x.shape[0]).T
115             Z[j] = multivariate_gaussian(points, mu[i], covar[i])
116         plt.contour(X, Y, Z, 1)
117
118     # plot the samples
119     plt.plot(x_data, y_data, 'co', zorder=1)
120     plt.xlabel('x')
121     plt.ylabel('y')
122     plt.title("Mixtures after {} steps".format(iteration + 1))
123
124
125 def calculate_log_likelihood(x, mu, covar, pi):
126     likelihood = np.empty((k, x.shape[0]))
127     for i in range(k):
128         likelihood[i] = pi[i] * multivariate_gaussian(x, mu[i], covar[i])
129
130     return np.sum(np.log(np.sum(likelihood, axis=0)))
131
132
133 if __name__ == '__main__':
134     main()
```

Plots at different iteration $t_i \in 1, 3, 5, 10, 30$ are generated for showing the data and the mixture component. The

log-likelihood for every iteration is plotted as well.

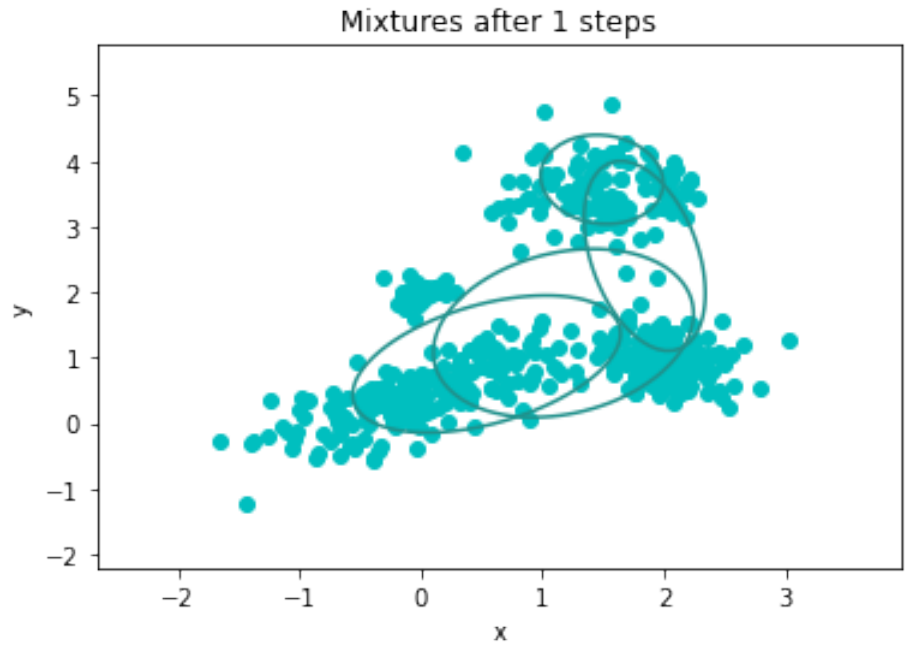

Figure 13

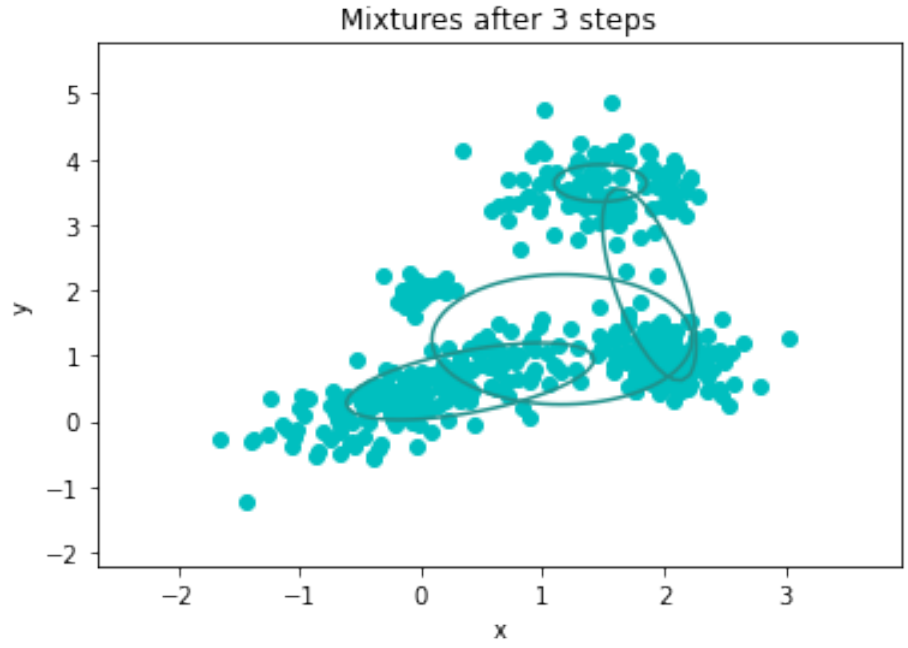

Figure 14

It is obvious that the algorithm has already to cluster all data points in the 30th iteration. Besides that, according to the log-likelihood, there is no improvement in the estimation after the 8th iteration so that the stable state has already achieved.

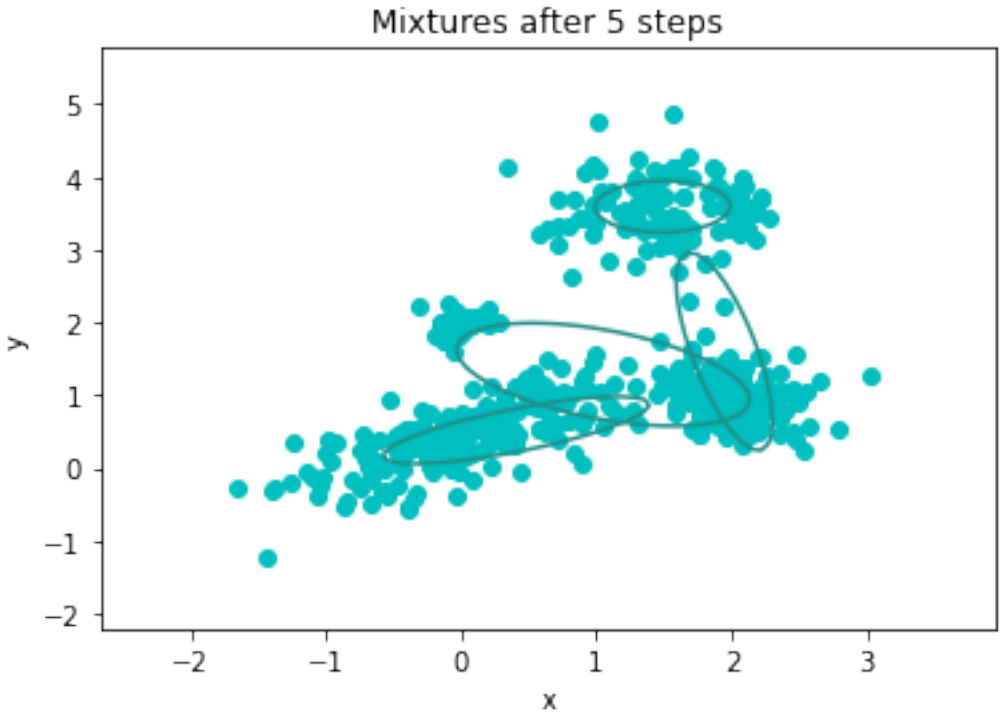

Figure 15

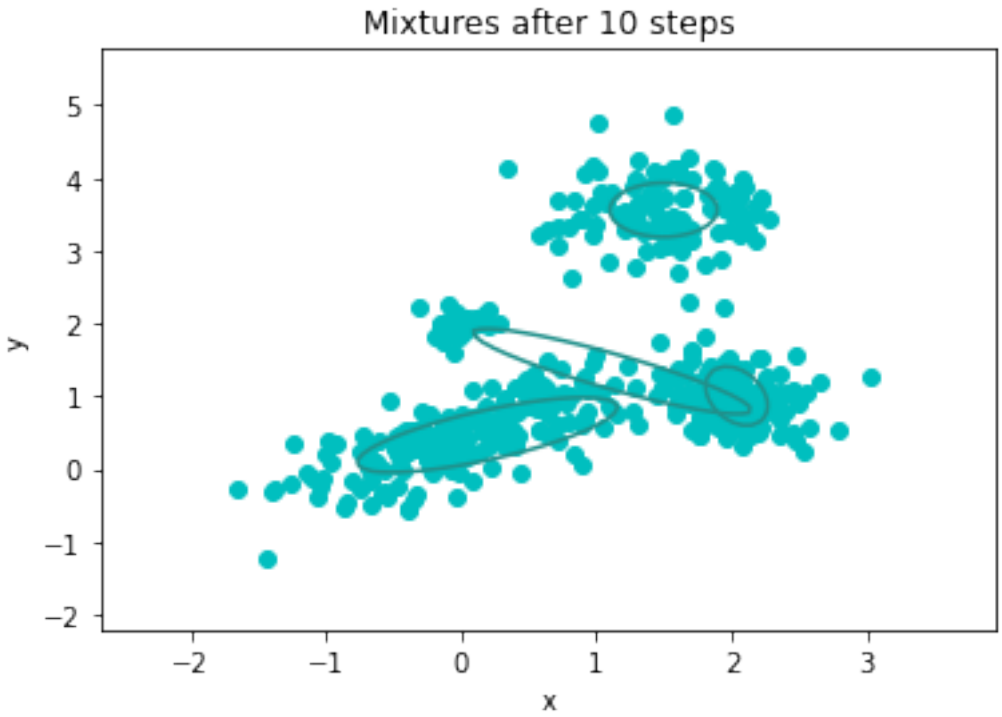

Figure 16

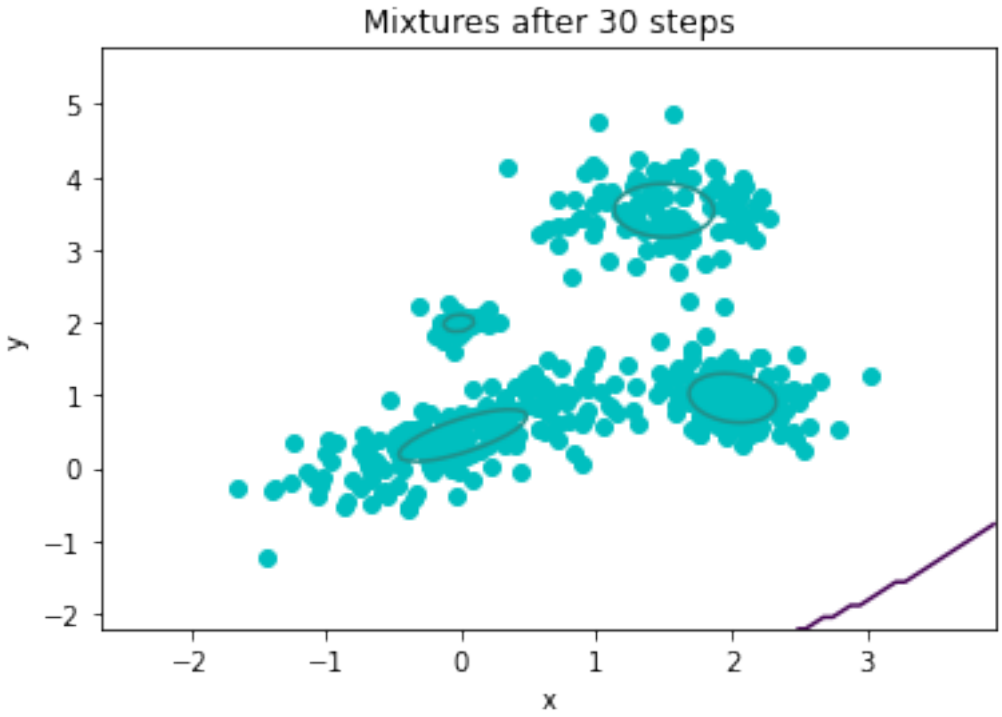

Figure 17

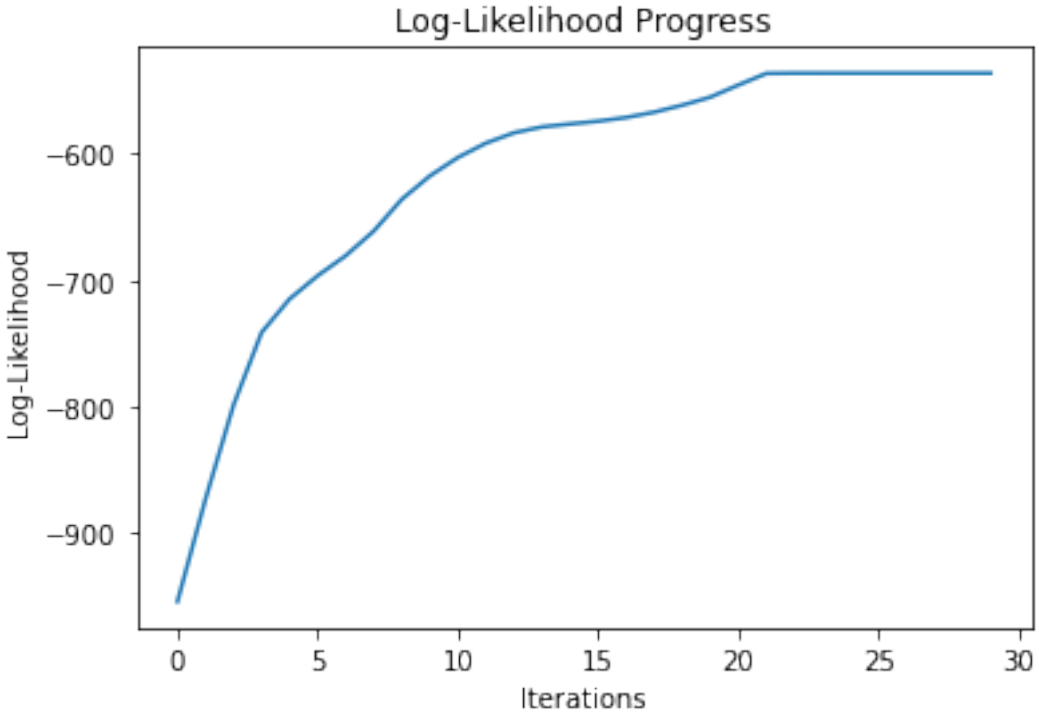

Figure 18

References

[1] Bhavesh Bhatt. *Stochastic vs Batch vs Mini-Batch Gradient Descent*. URL: https://www.youtube.com/watch?v=Ne3hjpP7KSI.

[2] G-kdom. *SGD with Momentum, SGD with Nesterov Momentum*. URL: https://zhuanlan.zhihu.com/p/73264637.

[3] dProgrammer lopez. *Mini-batch, Stochastic Batch Gradient Descent | Tutorial*. URL: https://www.youtube.com/watch?v=EAZs9qnR6mM.