Reproduction and Promotion of Single-cell RNA-seq denoising using a deep count autoencoder

Experiment Description

Denoise single-cell RNA-seq data. Single-cell RNA's RNA-seq is measured in biology experiments to analyse gene expression. Some RNA-seq cannot be accurately measured and is recorded as 0. However, there are also some genes actually do not express and their real value is 0. The work done in the original paper is using DCA to denoise single-cell RNA-seq data, that is, to judge whether the record 0 is generated from measurement or lack of expression, and to estimate the real value and fill in the appropriate position.

Experiment Requirements

- 基本要求

  将test_data.csv,test_truedata.csv分为测试集和验证集。实现任意补插算法来对数据集data.csv进行补插。使用测试集确定一个较好的模型，并使用验证集验证。针对你的模型和实验写一份报告，并提交源码(说明运行环境)和可执行文件。(建议使用神经网络)

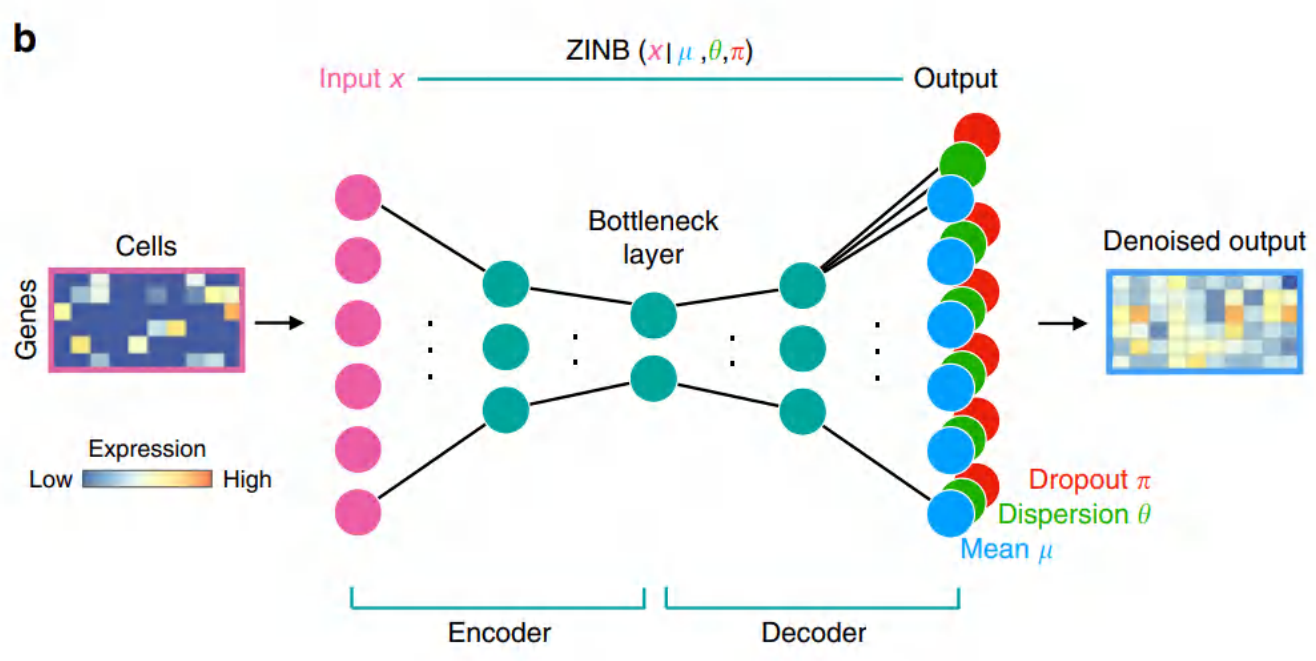
- 中级要求

  在test_data.csv上获得一定效果(dropout率变小，表达差异变小) 。在别人理论的基础上进行一些自己的修改。

- 高级要求

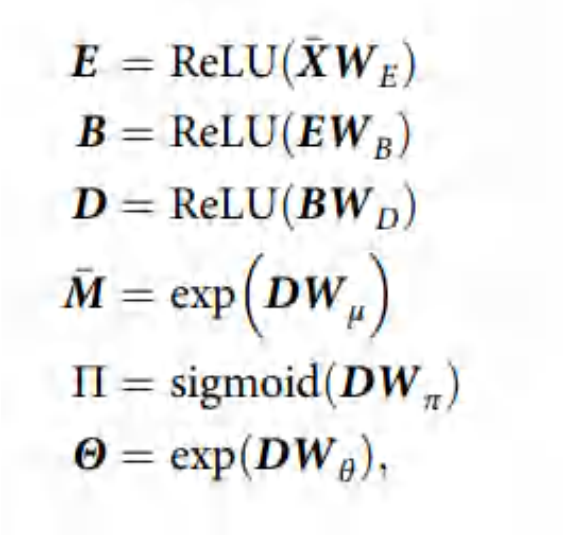
  对于data.csv的补插和生成data.csv的模拟真实数据集相比获得较好效果(dropout率变小，表达差异变小) 。

Experiment Principle



Model Construction

```
class AutoEncoder(nn.Module):
```



$$
\begin{aligned}
\boldsymbol{E} &= \mathrm{ReLU}(\bar{\boldsymbol{X}}\boldsymbol{W}_E) \\
\boldsymbol{B} &= \mathrm{ReLU}(\boldsymbol{E}\boldsymbol{W}_B) \\
\boldsymbol{D} &= \mathrm{ReLU}(\boldsymbol{B}\boldsymbol{W}_D) \\
\bar{\boldsymbol{M}} &= \exp\left(\boldsymbol{D}\boldsymbol{W}_\mu\right) \\
\Pi &= \mathrm{sigmoid}(\boldsymbol{D}\boldsymbol{W}_\pi) \\
\boldsymbol{\Theta} &= \exp(\boldsymbol{D}\boldsymbol{W}_\theta),
\end{aligned}
$$

- Input: count matrix A

  A_{ij} represents the expression the i gene in the j cell.

- Hidden Layer:

  E, B, D in the picture above reprensent encoding layers, bottleneck layer and decoding layers.

  All hidden layers except the bottleneck layer includes 64 neurons, and the bottleneck layer has 32 neurons.

- Output: The three parameters of every gene (μ, θ, π), that is (mean, dispersion, the probability of dropout), and is also the three parameters of ZINB distribution.

Specify the numbers of neurons per layer

```
def __init__(self, input_size=None, hasBN=False):

        super().__init__()
        self.intput_size = input_size
        self.hasBN=hasBN
        self.input = Linear(input_size, 64)
        self.bn1 = BatchNorm1d(64)
        self.encode = Linear(64, 32)
        self.bn2 = BatchNorm1d(32)
        self.decode = Linear(32, 64)
        self.bn3 = BatchNorm1d(64)
        self.out_put_PI = Linear(64, input_size)
        self.out_put_M = Linear(64, input_size)
        self.out_put_THETA = Linear(64, input_size)
        self.reLu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
```

Specify the activation function

The activation function of THETA and M are both exp, this can make sure the output is legitimate because they are both non-negative values.

Use Sigmoid as the activation function, because it is the estimated dropout probability and it should be between 0 and 1.

```
def forward(self, x):
        x = self.input(x)
        if self.hasBN:x = self.bn1(x)
        x = self.reLu(x)
        x = self.encode(x)
        if self.hasBN:x = self.bn2(x)
        x = self.reLu(x)
        x = self.decode(x)
        if self.hasBN:x = self.bn3(x)
        x = self.reLu(x)
        PI = self.sigmoid(self.out_put_PI(x))
        M = torch.exp(self.out_put_M(x))
        THETA = torch.exp(self.out_put_THETA(x))
        return PI, M, THETA
```

Define the loss function of the model

$$\hat{\Pi}, \hat{\boldsymbol{M}}, \hat{\Theta} = \operatorname{argmin}_{\Pi,\boldsymbol{M},\boldsymbol{\Theta}} \mathrm{NLL}_{\mathrm{ZINB}}(\boldsymbol{X}; \Pi, \boldsymbol{M}, \boldsymbol{\Theta}) + \lambda \|\Pi\|_F^2$$

$$= \operatorname{argmin}_{\Pi,\boldsymbol{M},\boldsymbol{\Theta}} \sum_{i=1}^{n} \sum_{j=1}^{p} \mathrm{NLL}_{\mathrm{ZINB}}(x_{ij}; \pi_{ij}, \mu_{ij}, \theta_{ij}) + \lambda \pi_{ij}^2,$$

NLL is the negative logarithmic likelihood, and the ZINB formula is as below:

$$\mathrm{NB}(x; \mu, \theta) = \frac{\Gamma(x+\theta)}{\Gamma(\theta)} \left(\frac{\theta}{\theta+\mu}\right)^{\theta} \left(\frac{\mu}{\theta+\mu}\right)^{x}$$

$$\mathrm{ZINB}(x; \pi, \mu, \theta) = \pi\delta_0(x) + (1-\pi)\mathrm{NB}(x; \mu, \theta)$$

The ZINB distribution is chosen is because scRNA-seq data has many 0 values, and whose negative logarithmic likelihood is used as the loss function. When the loss function is minimum, the probability and the liklihood of ZINB are greatest. The training to be conducted is to reduce the loss and get the best estimation of ZINB parameters. Then use the estimated ZINB function to generate interpolation values.

Loss function definiton:

```
eps = torch.tensor(1e-10)
        THETA = torch.minimum(THETA, torch.tensor(1e6))
        t1 = torch.lgamma(THETA + eps) + torch.lgamma(X + 1.0) - torch.lgamma(X + THETA +
eps)
        t2 = (THETA + X) * torch.log(1.0 + (M / (THETA + eps))) + (X * (torch.log(THETA +
eps) - torch.log(M + eps)))
        nb = t1 + t2
        nb = torch.where(torch.isnan(nb), torch.zeros_like(nb) + max1, nb)
        nb_case = nb - torch.log(1.0 - PI + eps)
        zero_nb = torch.pow(THETA / (THETA + M + eps), THETA)
        zero_case = -torch.log(PI + ((1.0 - PI) * zero_nb) + eps)
        res = torch.where(torch.less(X, 1e-8), zero_case, nb_case)
        res = torch.where(torch.isnan(res), torch.zeros_like(res) + max1, res)
        return torch.mean(res)
```

Input

The input function is as shown below. X is original counting matrix, and S_i is the factor of proportion of every cell.

$$\overline{X} = zscore(log(diag(s_i)^{-1}X + 1))$$

Because NLL estimation has been used, we need to preprocess infinite and invalid values in original dataset. Implement z-score normalization, that is, subtract the mean from the data and divide it by its standard deviation. The outcome of the normalization is that for every attribute, all data is clustered around 0 and is with a deviation of 1. The formula is: $x^* = \frac{x-\overline{x}}{\sigma}$

```
def preprocess_data(data: Tensor):
    gene_num = data.shape[1]
    s = torch.kthvalue(data,gene_num//2,1)
    s = 1/s.values
    norm_data = torch.matmul(torch.diag(s), data) + 1
    norm_data = torch.log(norm_data)
    norm_data = (norm_data - norm_data.mean()) / norm_data.std()
    return norm_data
```

Training

Use autoencoder for training, use batchsize = 32, that is, use the data of 32 cells for training each time.

```
def train(EPOCH_NUM=100, print_batchloss=False, autoencoder=None, loader=None,
startEpoch=0):
    """

    :param print_batchloss: if print train infor, False by default
    """
    opt = Adam(autoencoder.parameters(), lr=LR, betas=(BETA1, BETA2), eps=EPS,
weight_decay=WEIGHT_DECAY)
```

```
    # opt = SGD(autoencoder.parameters(), lr=1e-2, momentum=0.8)
    mean_loss=0
    for epoch in range(EPOCH_NUM+1):
        epoch_loss = 0

        for batch, batch_data in enumerate(loader):
            # batchsize = 32

            opt.zero_grad()
            train_batch = batch_data[0]

            # d: original matrix
            d = train_batch[:, :, 0]
            # norm_d: proessed data
            norm_d = train_batch[:, :, 1]
```

Forward and calculate loss value.

Follow the sequence of input to output, calculate and store intermediate variables of the model to train the next layer.

```
# forward
            PI, M, THETA = autoencoder(norm_d)
            templ = lzinbloss(d, PI, M, THETA)
            epoch_loss += templ
            if print_batchloss:
                print(f'epoch:{epoch+startEpoch},batch:{batch},batch loss:{templ},(batch
size {BATCHSIZE})')
                f.write(f'epoch:{epoch+startEpoch},batch:{batch},batch loss:{templ},(batch
size {BATCHSIZE})\n')

            # gradient descent
            opt.step()
        mean_loss+=epoch_loss
        if epoch % 100 ==0:
            mean_loss=mean_loss/100
            print(f'epoch:{epoch+startEpoch},epoch loss:{mean_loss}')
            f.write(f'epoch:{epoch+startEpoch},epoch loss:{mean_loss}\n')
            mean_loss=0
        # if epoch % EVER_SAVING == 0 and epoch!=0: torch.save(autoencoder.state_dict(),
open(f'0113epoch{epoch+startEpoch}.pkl', 'wb'))
        if epoch % EVER_SAVING == 0 and epoch!= 0 : torch.save(autoencoder.state_dict(),
open(f'0113epoch{epoch+startEpoch}withoutBN.pkl', 'wb'))
```

Compute gradients with backpropagation

```
            templ.backward()
            clip_grad_norm_(autoencoder.parameters(), max_norm=5, norm_type=2)
```

Considering the output PI,M,THETA, if the gene's corresponding PI>0.5, fill with corresponding M value as below:

```
autoencoder.load_state_dict(torch.load(STATE_DICT_FILE))
    print(autoencoder)
    PI, M, THETA = autoencoder(norm_data)
    iszero = data == 0
    predict_dropout_of_all = PI>0.5
    # dropout_predict = torch.where(predict_mask, M, torch.zeros_like(PI))
    # print("after",after)

    true_drop_out_mask = iszero*((truedata - data)!=0)
    predict_dropout_mask = iszero*predict_dropout_of_all
    after = torch.floor(torch.where(predict_dropout_mask,M,data))
    zero_num = iszero.sum()
    true_dropout_num = true_drop_out_mask.sum()
    predict_dropout_num = predict_dropout_mask.sum()
    print("predict_dropout_num:",predict_dropout_num,
          "\ntrue_dropout_num:", true_dropout_num,
          "\nzero_num:",zero_num,
          "\npredict out of true dropout rate:",
(predict_dropout_mask*true_drop_out_mask).sum()/true_dropout_num)

    dif_after =  truedata - after
    dif_true = truedata - data
    # print(dif_after)
    # print(dif_true)
    print("predict distance:", torch.sqrt(torch.square(truedata - after).sum()).data,
          "origin distance:", torch.sqrt(torch.square(truedata - data).sum()).data)
```

Analysis and Result Display

Split *test_data.csv*,*test_truedata.csv* into test set and validation set in a ratio of 7:3.

Test four models on test dataset

hasBN: whether has BatchNorm1d layer

zinb_new: whether use the improved zinb function

Model I：hasBN=False zinb_new=False

Gradient explosion occurred in the later stage of training.

```
epoch:2473,epoch loss:25700.482421875
epoch:2474,epoch loss:25913.857421875
epoch:2475,epoch loss:25945.88671875
epoch:2476,epoch loss:26105.73046875
epoch:2477,epoch loss:nan
epoch:2478,epoch loss:665005.1875
epoch:2479,epoch loss:665005.25
epoch:2480,epoch loss:665005.1875
epoch:2481,epoch loss:665005.25
epoch:2482,epoch loss:665005.25
```

1000 epochs:

```
predict_dropout_num: tensor(10385)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(1463)
predict out of true dropout rate: tensor(0.1066)
precision: tensor(0.1409)
recall: tensor(0.1066)
predict distance: tensor(10595932.) origin distance: tensor(2709.5833)
```

Use Euclidean distance to represent the difference of predict value and real value of data. The difference is tremendous, and it is clearly underfitting.

2000 epochs:

```
predict_dropout_num: tensor(10914)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(1770)
predict out of true dropout rate: tensor(0.1290)
precision: tensor(0.1622)
recall: tensor(0.1290)
predict distance: tensor(2.1165e+13) origin distance: tensor(2709.5833)
```

Still underfitting.

3000 epochs:

```
predict_dropout_num: tensor(0)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(0)
predict out of true dropout rate: tensor(0.)
precision: tensor(nan)
recall: tensor(0.)
predict distance: tensor(2709.5833) origin distance: tensor(2709.5833)
```

Gradient explosion.

Model II：hasBN=True zinb_new=False

```
epoch:0,epoch loss:104700.0390625
epoch:1,epoch loss:99823.984375
epoch:2,epoch loss:95683.0234375
epoch:3,epoch loss:92058.03125
epoch:4,epoch loss:88519.203125
epoch:5,epoch loss:85298.8359375
epoch:6,epoch loss:82156.0625
epoch:7,epoch loss:79108.2578125
epoch:8,epoch loss:76354.2734375
epoch:9,epoch loss:73663.265625
epoch:10,epoch loss:71114.0390625
epoch:11,epoch loss:68649.171875
```

```
epoch:1687,epoch loss:11143.952148437
epoch:1688,epoch loss:11155.482421875
epoch:1689,epoch loss:11211.53515625
epoch:1690,epoch loss:11219.3134765625
epoch:1691,epoch loss:11311.5048828125
epoch:1692,epoch loss:11085.9189453125
epoch:1693,epoch loss:11236.861328125
epoch:1694,epoch loss:11309.673828125
epoch:1695,epoch loss:11510.396484375
epoch:1696,epoch loss:11509.431640625
epoch:1697,epoch loss:11127.2431640625
epoch:1698,epoch loss:11108.359375
epoch:1699,epoch loss:11152.9609375
```

1000 epochs:

```
predict_dropout_num: tensor(16365)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(2796)
predict out of true dropout rate: tensor(0.2038)
precision: tensor(0.1709)
recall: tensor(0.2038)
predict distance: tensor(2715.3228) origin distance: tensor(2709.5833)
```

2000 epochs:

```
predict_dropout_num: tensor(16997)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(3116)
predict out of true dropout rate: tensor(0.2271)
precision: tensor(0.1833)
recall: tensor(0.2271)
predict distance: tensor(2744.3308) origin distance: tensor(2709.5833)
```

3000 epochs:

```
predict_dropout_num: tensor(17500)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(3297)
predict out of true dropout rate: tensor(0.2403)
precision: tensor(0.1884)
recall: tensor(0.2403)
predict distance: tensor(2758.3293) origin distance: tensor(2709.5833)
```

Model III：hasBN=False zinb_new=True

3000 epochs:

```
predict_dropout_num: tensor(17950)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(5515)
predict out of true dropout rate: tensor(0.4020)
precision: tensor(0.3072)
recall: tensor(0.4020)
predict distance: tensor(6.7696e+17) origin distance: tensor(2709.5833)
```

Model IV：hasBN=False zinb_new=True

```
epoch:0,epoch loss:227.91006469726562
epoch:1,epoch loss:203.54388427734375
epoch:2,epoch loss:186.39576721191406
epoch:3,epoch loss:170.11827087402344
epoch:4,epoch loss:154.9748992919922
epoch:5,epoch loss:139.35003662109375
epoch:6,epoch loss:125.00446319580078
epoch:7,epoch loss:114.38752746582031
epoch:8,epoch loss:102.25062561035156
epoch:9,epoch loss:93.86244201660156
epoch:10,epoch loss:84.02640533447266
epoch:11,epoch loss:75.88536834716797
epoch:1028,epoch loss:13.597317895817878
epoch:1029,epoch loss:13.537382125854492
epoch:1030,epoch loss:13.36414909362793
epoch:1031,epoch loss:13.37580394744873
epoch:1032,epoch loss:13.530522346496582
epoch:1033,epoch loss:13.511974334716797
epoch:1034,epoch loss:13.40842056274414
epoch:1035,epoch loss:13.496503829956055
epoch:1036,epoch loss:13.493284225463867
epoch:1037,epoch loss:13.347700119018555
epoch:1038,epoch loss:13.425634384155273
```

1000 epochs:

```
predict_dropout_num: tensor(23635)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(7735)
predict out of true dropout rate: tensor(0.5638)
precision: tensor(0.3273)
recall: tensor(0.5638)
predict distance: tensor(1518.7639) origin distance: tensor(2709.5833)
```

2000 epochs:

```
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(9373)
predict out of true dropout rate: tensor(0.6832)
precision: tensor(0.3704)
recall: tensor(0.6832)
predict distance: tensor(1102.5829) origin distance: tensor(2709.5833)
```

3000 epochs:

```
predict_dropout_num: tensor(25872)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(10035)
predict out of true dropout rate: tensor(0.7314)
precision: tensor(0.3879)
recall: tensor(0.7314)
predict distance: tensor(983.3555) origin distance: tensor(2709.5833)
```

Conclusion: Model IV has the best performance. The new zinb loss function is better, and using batchNorm1d layer is better than not using it.

Validate on the validation set

Predict on the validation set using 3000 epoch model.

Model I:

Model I has poor performance, and gradient explode after 3000 epoches, so it is not comparable. The outcome is not listed here.

Model II:

```
predict_dropout_num: tensor(8715)
true_dropout_num: tensor(5824)
zero_num: tensor(14210)
predict_correct: tensor(1766)
predict out of true dropout rate: tensor(0.3032)
precision: tensor(0.2026)
recall: tensor(0.3032)
predict distance: tensor(2125.3137) origin distance: tensor(2123.7241)
```

Model III:

```
predict_dropout_num: tensor(8889)
true_dropout_num: tensor(5824)
zero_num: tensor(14210)
predict_correct: tensor(2997)
predict out of true dropout rate: tensor(0.5146)
precision: tensor(0.3372)
recall: tensor(0.5146)
predict distance: tensor(1.9116e+08) origin distance: tensor(2123.7241)
```

Model IV:

```
predict_dropout_num: tensor(13086)
true_dropout_num: tensor(5824)
zero_num: tensor(14210)
predict_correct: tensor(5333)
predict out of true dropout rate: tensor(0.9157)
precision: tensor(0.4075)
recall: tensor(0.9157)
predict distance: tensor(772.4746) origin distance: tensor(2123.7241)
```

It can be seen that model IV has the best performance, with high precision and recall, and the lowest predict distance.

Analyzation

Model I is the original model reproduced as described in the paper. However, the performance of Model I is undesirable, and the speed of its gradient descent is slow. Besides, after certain epoches, the loss is NA, which represents calculation error or numerical anormaly. This can be caused by gradient explosion, which makes the parameters to be extreme and leads to overflow, making the value be inf, and result in NaN eventually. Thus the batch normalization layer is introduced into the neural network, that is, set `autoencoder = AutoEncoder(1000,hasBN=True)` , and test its performance. It can be seen that the denoise of data.csv has a fair performance compared with the simulated real dataset that generated data.csv, with less dropout and less difference.

After reading the whole paper, it is found that some experiment result cannot be reproduced according to the method described in the paper. Besides, the source code is different with the text description. Thus, re-implement loss calculation and training technique as below:

```
class LZINBLoss(nn.Module):
    def __init__(self, eps=1e-6):
        super().__init__()
        self.eps = eps

    def forward(self, X: Tensor, PI: Tensor = None, M: Tensor = None, THETA: Tensor =
None):
        # in case of log(0) log (neg num)
        eps = self.eps
        # deal with inf
        max1 = max(THETA.max(), M.max())
        if THETA.isinf().sum() != 0:
            THETA = torch.where(THETA.isinf(), torch.full_like(THETA, max1), THETA)
        if M.isinf().sum() != 0:
            M = torch.where(M.isinf(), torch.full_like(M, max1), M)
```

```
        if PI.isnan().sum() != 0:
            PI = torch.where(PI.isnan(), torch.full_like(PI, eps), PI)
        if THETA.isnan().sum() != 0:
            THETA = torch.where(THETA.isnan(), torch.full_like(THETA, eps), THETA)
        if M.isnan().sum() != 0:
            M = torch.where(M.isnan(), torch.full_like(M, eps), M)

        eps = torch.tensor(1e-10)
        THETA = torch.minimum(THETA, torch.tensor(1e6))
        t1 = torch.lgamma(THETA + eps) + torch.lgamma(X + 1.0) - torch.lgamma(X + THETA +
eps)
        t2 = (THETA + X) * torch.log(1.0 + (M / (THETA + eps))) + (X * (torch.log(THETA +
eps) - torch.log(M + eps)))
        nb = t1 + t2
        nb = torch.where(torch.isnan(nb), torch.zeros_like(nb) + max1, nb)
        nb_case = nb - torch.log(1.0 - PI + eps)
        zero_nb = torch.pow(THETA / (THETA + M + eps), THETA)
        zero_case = -torch.log(PI + ((1.0 - PI) * zero_nb) + eps)
        res = torch.where(torch.less(X, 1e-8), zero_case, nb_case)
        res = torch.where(torch.isnan(res), torch.zeros_like(res) + max1, res)
        return torch.mean(res)
```

```
def train(EPOCH_NUM=100, print_batchloss=False, autoencoder=None, loader=None,
startEpoch=0):
    opt = Adam(autoencoder.parameters(), lr=LR, betas=(BETA1, BETA2), eps=EPS,
weight_decay=WEIGHT_DECAY)
    # opt = SGD(autoencoder.parameters(), lr=1e-2, momentum=0.8)
    mean_loss=0
    for epoch in range(EPOCH_NUM+1):
        epoch_loss = 0

        for batch, batch_data in enumerate(loader):
            opt.zero_grad()
            train_batch = batch_data[0]
            d = train_batch
            PI, M, THETA = autoencoder(d)
            templ = lzinbloss(d, PI, M, THETA)
            epoch_loss += templ
            if print_batchloss:
                print(f'epoch:{epoch+startEpoch},batch:{batch},batch loss:{templ},(batch
size {BATCHSIZE})')
                f.write(f'epoch:{epoch+startEpoch},batch:{batch},batch loss:{templ},(batch
size {BATCHSIZE})\n')
            # backpropagtion
            templ.backward()
            clip_grad_norm_(autoencoder.parameters(), max_norm=5, norm_type=2)
            # gra descent
            opt.step()
        print(f'epoch:{epoch+startEpoch},epoch loss:{epoch_loss}')
        f.write(f'epoch:{epoch+startEpoch},epoch loss:{mean_loss}\n')
```

```
        # mean_loss=0
        if epoch % EVER_SAVING == 0 and epoch!=0: torch.save(autoencoder.state_dict(),
open(f'0113epoch{epoch+startEpoch}.pkl', 'wb'))
        # if epoch % EVER_SAVING == 0 and epoch!= 0 : torch.save(autoencoder.state_dict(),
open(f'0113epoch{epoch+startEpoch}withoutBN.pkl', 'wb'))
```

After the improvement, test on the test set, the result is shown as below, and the denoised data is saved in result.csv.

```
epoch:2984,epoch loss:477.8699035644531
epoch:2985,epoch loss:477.9356384277344
epoch:2986,epoch loss:477.7810363769531
epoch:2987,epoch loss:477.9073791503906
epoch:2988,epoch loss:477.40167236328125
epoch:2989,epoch loss:477.8121337890625
epoch:2990,epoch loss:477.7042541503906
epoch:2991,epoch loss:477.810302734375
epoch:2992,epoch loss:477.6019287109375
epoch:2993,epoch loss:477.909423828125
epoch:2994,epoch loss:477.867919921875
epoch:2995,epoch loss:478.1471252441406
epoch:2996,epoch loss:477.696044921875
epoch:2997,epoch loss:477.8654479980469
epoch:2998,epoch loss:477.6917724609375
epoch:2999,epoch loss:478.10479736328125
epoch:3000,epoch loss:477.5592041015625
```

```
AutoEncoder(
  (input): Linear(in_features=1000, out_features=64, bias=True)
  (bn1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (encode): Linear(in_features=64, out_features=32, bias=True)
  (bn2): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (decode): Linear(in_features=32, out_features=64, bias=True)
  (bn3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (out_put_PI): Linear(in_features=64, out_features=1000, bias=True)
  (out_put_M): Linear(in_features=64, out_features=1000, bias=True)
  (out_put_THETA): Linear(in_features=64, out_features=1000, bias=True)
  (reLu): ReLU()
  (sigmoid): Sigmoid()
)
predict_dropout_num: tensor(619364)
zero_num: tensor(1150882)
```

Decision Tree

Experiment Description

基本要求

1. 基于 Watermelon-train1数据集（只有离散属性），构造ID3决策树；
2. 基于构造的 ID3 决策树，对数据集 Watermelon-test1进行预测，输出分类精度；

中级要求

1. 对数据集Watermelon-train2，构造C4.5或者CART决策树，要求可以处理连续型属性；
2. 对测试集Watermelon-test2进行预测，输出分类精度；

Data Import

```
import numpy as np
import pandas as pd
import math
import copy
def readfile(name):
    df = pd.read_csv(name, error_bad_lines = False, encoding = 'gbk')
    temp = np.array(df).tolist()
    for i in temp:
        i.pop(0)
    return temp

train1 = readfile("Watermelon-train1.csv")
train2 = readfile("Watermelon-train2.csv")
test1 = readfile("Watermelon-test1.csv")
test2 = readfile("Watermelon-test2.csv")
```

Calculate Information Gain

- Count the frequency of every single samples, and store the data in a dictionary.
- Calculate information entropy
- Return information entropy and classification information

```
def information(data):
    dic = {}
    for i in data:
        current = i[-1] #最后结果
        if current not in dic.keys():
            dic[current] = 1 #新类别
 else:
            dic[current] += 1 #原有类别+1
    result = 0.0
    for key in dic:
        prob = float(dic[key]) / len(data)
        result -= prob * math.log(prob,2)
    return result, dic
```

- Classify samples according to an feature, and delete the feature.

```
def split(data, index, kind):
    ls = []
    for temp in data:
        if temp[index] == kind:
            t = temp[0: index]
            t = t + temp[index + 1: ]
            ls.append(t)
    return ls
```

- Calculate information entropy and information gain. Return the best classification method, and conduct the process below:
    1. Extract all possible values of the feature.
    2. Split the dataset into multiple subsets.
    3. Calculate the information entropy of every subsets.
    4. Calculate the information gain.
    5. Choose the feature with the most information gain and return the index of the feature.

```
def chooseBestFeatureToSplit(data):
    base, mm= information(data)
    best = 0
    bestindex = -1
    for i in range(len(data[0]) - 1):
        ls = [index[i] for index in data]
        feture = set(ls)
```

```
        temp = 0.0
        for value in feture:
            datatemp = split(data, i, value)
            prob = len(datatemp) / float(len(data))
            t, mm = information(datatemp)
            temp += prob * t
        infoGain = base - temp
        if infoGain > best:
            best = infoGain
            bestindex = i
    return bestindex
```

Recursively Build ID3 Decision Tree

Generate Decision Tree

1. Calculate the information gain of all features from the root node, and choose the feature with most information gain to split the node. Create child node according to different values of the feature.
2. Invoke the method above recuisively on child nodes until reach the stop condition and generate a decision tree.

Stop Condition for Recursion

1. All samples of current node belongs to the same set.
2. The current node has single values in all features and has no other feature for future split.
3. Reach the maximum depth of the tree.
4. Reach the minimum sample size of leaf node.

Implementation

1. Extract the types of the dataset.
2. Calculate values and numbers of types. If the dataset only has one type, return the type.
3. Calculate the index of the feature for the best split.
4. Initialize the subtree.
5. Delete the feature has been used for classification to avoid reuse.
6. Calculate remaining features.
7. Implement recursion on every branch.

8. Return subtree.

```
def classify1(data, labels):
    typelist = [index[-1] for index in data]
    nothing, typecount = information(data)
    if len(typecount) == 1:
        return typelist[0]
    bestindex = chooseBestFeatureToSplit(data)
    bestlabel = labels[bestindex]
    Tree = {bestlabel: {}}
    temp = labels[:]
    del (temp[bestindex])
    feature = [example[bestindex] for example in data]
    unique = set(feature)
    for i in unique:
        temp = temp[:]
        Tree[bestlabel][i] = classify1(split(data, bestindex, i), temp)
    return Tree
```

Classify Test Dataset Using ID3 Decision Tree

- Extract the root node of the tree.
- Compare input data with the value of root keys.
- Iterate through the keys. If the value of the equivalent key is a value, then it is a leaf node, return the value.
- If it is not a leaf node, traversally query the subtree.

```
def run1(testdata, tree, labels):
    firstStr = list(tree.keys())[0]
    secondDict = tree[firstStr]
    featIndex = labels.index(firstStr)
    result = ''
    for key in list(secondDict.keys()):
         if testdata[featIndex] == key:
            if type(secondDict[key]).__name__ == 'dict':  # 不是叶子节点，递归

                result = run1(testdata, secondDict[key], labels)
            else:
                result = secondDict[key]
    return result
```

```
生成决策树如下: {'纹理': {'稍糊': {'色泽': {'乌黑': {'敲声': {'浊响': '是', '沉闷': '否'}}, '浅白': '否', '青绿': '否'}}, '清晰': {'根蒂': {'硬挺': '否', '蜷缩': '是', '稍蜷': '是'}}, '模糊': '否'}}
```

Calculate Accuracy

```
def simrate(data, predict):
    num = 0
    for i in range(len(data)):
        if data[i][-1] == predict[i]:
            num +=1
    return format(num / len(data), '.2%')
print("ID3分类器对于test1数据集的准确率是：", simrate(test1, result1))
```

```
ID3分类器对于test1数据集的准确率是：  70.00%
```

It can be seen that, this method has a relatively good result on small dataset.

Build Decision Tree Using $C4.5$ Algorithm

The problem of using information gain as classificaiton critiria:

Information gain tend to choose the feature with more distinct values. For example, every student has a student ID. Thus, if we categorize based on that, every student belongs to a different group, which is meaningless. However, C4.5 Algorithm use information gain ratio to choose features used for classification. This can solve the problem.

Information gain ratio = penalty parameter * information gain: $g_R(D,A) = \frac{g(D,A)}{H_A(D)}. H_A(D)$ uses current feature A as random variable in sample set D to get the empirical entropy.

The essence of information gain ratio is to mutiply information gain with a penalty parameter. When there are more feature values, the panalty parameter is smaller; when there are less feature values, the panalty parameter is bigger.

Panalty parameter: the reciprocal of the entropy of A as random variable in dataset D. That is, put samples with the same feature A values in the same group, and the panalty parameter is $\frac{1}{H_A(D)} = \frac{1}{-\sum_{i=1}^{n} \frac{|D_i|}{|D|} log_2 \frac{|D_i|}{|D|}}$

- Discretization of continuous features

  Use Binary Splitting to arrange data from smallest to largest, and choose the midpoint value for each interval. The left interval is the value smaller than the midpoint value, and the right interval is the value bigger than the midpoint value.

```
def Division(data):
    ls = data[:]
    ls.sort()
    result = []
    for i in range(len(ls) - 1):
        result.append((data[i + 1] + data[i]) / 2)
    return result
```

- Reclassify data base on the index of a feature: method = 0 put data not greater than the midpoint to the left; method = 1 put data greater than the midpoint to the right.

```
def split2(data, index, kind, method):
    ls = []
    if method == 0:
        for temp in data:
            if temp[index] <= kind:
                t = temp[0 : index]
                t = t + temp[index + 1 : ]
                ls.append(t)
    else:
        for temp in data:
            if temp[index] > kind:
                t = temp[0 : index]
                t = t + temp[index + 1 : ]
                ls.append(t)
    return ls
```

  - Calculate information entropy and informatin gain and return the best classification method. Conduct the following process on every feature of the dataset:
    1. Extract all possible values of the feature.
    2. Calculate the information entropy of every feature.
    3. Split the dataset into multiple subsets based on feature values.
    4. Calculate the information entropy of every subsets.
    5. Calculate the information gain ratio.
    6. Choose the feature with greatest information gain ratio and return the index of the feature.
    7. For continuous data, calculate the information gain ratio of every possible split point (midpoint of the intervals) and return the split point with least information gain ratio.

```
def chooseBestFeatureToSplit2(data):
    base, mm= information(data) #original infor gain
    info = []
    for j in range(len(data[0]) - 1):
        dic = {}
        for i in data:
            current = i[j] #extract result
            if current not in dic.keys():
                dic[current] = 1 #build new grp
            else:
                dic[current] += 1 #formal grp + 1
        result = 0.0
        for key in dic:
            prob = float(dic[key]) / len(data)
            result -= prob * math.log(prob,2)
        info.append(result)
    best = 0
    bestindex = -1
    bestpartvalue = None #use for discrete feature
    for i in range(len(data[0]) - 1):
        #extract all values of a feature
        ls = [index[i] for index in data]
        feture = set(ls)
        #cal infor gain
        temp = 0.0
        if type(ls[0]) == type("a"):#determine whether discrete
            for value in feture:
                datatemp = split(data, i, value)
                prob = len(datatemp) / float(len(data))
                t, mm = information(datatemp)
                temp += prob * t
        else:
            ls.sort()
            min = float("inf")
            for j in range(len(ls) - 1):
                part = (ls[j + 1] + ls[j]) / 2 #calculate split
point
                left = split2(data, i, part, 0)
                right = split2(data, i, part, 1)
                temp1, useless = information(left)
                temp2, useless = information(right)
                temp = len(left) / len(data) * temp1 + len(right) /
len(data) * temp2
                if temp < min:
                    min = temp
```

```
                    bestpartvalue = part
            temp = min
        infoGain = base - temp
        #choose by infor gain
        if info[i] != 0:
            if infoGain / info[i] >= best:
                best = infoGain / info[i]
                bestindex = i
    return bestindex, bestpartvalue
```

Build $C4.5$ Decision Tree Traversally

- Build the decision tree.
    1. Calculate the information gain of all features from the root node, and choose the feature with most information gain to split the node. Create child node according to different values of the feature.
    2. Invoke the method above recuisively on child nodes until reach the stop condition and generate a decision tree.
- Iteration stop condition.
    1. All samples of current node belongs to the same set.
    2. The current node has single values in all features and has no other feature for future split.
    3. Reach the maximum depth of the tree.
    4. Reach the minimum sample size of leaf node.
- Implementation.
    1. Extract the types of the dataset.
    2. Calculate values and numbers of types. If the dataset only has one type, return the type.
    3. Calculate the index of the feature for the best split.
    4. Initialize the subtree.
    5. Delete the feature has been used for classification to avoid reuse.
    6. Calculate remaining features.
    7. Implement recursion on every branch.
    8. Return subtree.

```
def classify2(data, labels):
    typelist = [index[-1] for index in data] #extract features
```

```
    nothing, typecount = information(data) #calculate infor gain
    if typecount == len(typelist): #if only 1 grp
        return typelist[0]
    bestindex, part = chooseBestFeatureToSplit2(data)  #the index of best
split feature
    if bestindex == -1:
        return "是"
    if type([t[bestindex] for t in data][0]) == type("a"):#if discrete
        bestlabel = labels[bestindex]
        Tree = {bestlabel: {}}
        temp = copy.copy(labels)
        feature = [example[bestindex] for example in data]
        del (temp[bestindex])  #chosen feature no longer used for
classification
        unique = set(feature)  #every possible values of the split feature
        for i in unique:
            s = temp[:]
            Tree[bestlabel][i] = classify2(split(data, bestindex, i), s)
    else: #continuous features
        bestlabel = labels[bestindex] + "<" + str(part)
        Tree = {bestlabel: {}}
        temp = labels[:]
        del(temp[bestindex])
        leftdata = split2(data, bestindex, part, 0)
        Tree[bestlabel]["是"] = classify2(leftdata, temp)
        rightdata = split2(data, bestindex, part, 1)
        Tree[bestlabel]["否"] = classify2(rightdata, temp)
    return Tree
```

Use $C4.5$ Decision Tree for Classification

- Extract the root node of the current tree.
- Use the root node to check the type of current data
- If it is discrete value:
    1. If the branch queried is a leaf node, return the value.
    2. If the branch queried is not a leaf node, traversally query the subtree.
- If it is non-discrete value:
    1. Extract the value to compare with current value.
    2. If the value is bigger then use "否" as label, else use"是" as label
    3. If the queried branch is a leaf node, return the value.
    4. If it is not leaf node, traversally query the subtree.

```
def run2(data, tree, labels):
    firstStr = list(tree.keys())[0]  # root node
    firstLabel = firstStr
    t = str(firstStr).find('<') # check whether feature continuous
    if t > -1:  # if continuous
        firstLabel = str(firstStr)[ : t]
    secondDict = tree[firstStr]
    featIndex = labels.index(firstLabel)  # node's feature
    result = ''
    for key in list(secondDict.keys()):  # iterate through branches
        if type(data[featIndex]) == type("a"):
            if data[featIndex] == key:
                if type(secondDict[key]).__name__ == 'dict':  # not leaf
node, traversal
                    result = run2(data, secondDict[key], labels)
                else:  # is leaf node, return
                    result = secondDict[key]
        else:
            value = float(str(firstStr)[t + 1 : ])
            if data[featIndex] <= value:
                if type(secondDict['是']).__name__ == 'dict':  # not leaf
node, traversal
                    result = run2(data, secondDict['是'], labels)
                else:  # is leaf node, return
                    result = secondDict['是']
            else:
                if type(secondDict['否']).__name__ == 'dict':  # not leaf
node, traversal
                    result = run2(data, secondDict['否'], labels)
                else:  # is leaf node, return
                    result = secondDict['否']
    return result

def getresult2(train, test, labels):
    ls = []
    tree = classify2(train, labels)
    print("生成决策树如下:", tree)
    for index in test:
        ls.append(run2(index, tree, labels))
    return ls

labels2 = ["色泽", "根蒂", "敲声", "纹理", "密度", "好瓜"]
result2 = getresult2(train2, test2, labels2)
```

```
生成决策树如下: {'纹理': {'稍糊': {'敲声': {'浊响': {'密度<0.56': {'是': '是', '否': '是'}}, '沉闷': {'密度<0.6615': {'是': '是', '否': {'根蒂': {'蜷缩': '是', '稍蜷': '是'}}}}}}, '清晰': {'根蒂': {'硬挺': '是', '蜷缩': {'密度<0.5820000000000001': {'是': '是', '否': {'敲声': {'浊响': {'色泽': {'乌黑': '是', '青绿': '是'}}}}}}, '沉闷': {'色泽': {'乌黑': '是', '青绿': '是'}}}}}}, '稍蜷': {'密度<0.3815': {'是': '是', '否': {'色泽': {'乌黑': '是', '青绿': '是'}}}}}}, '模糊': {'密度<0.29400000000000004': {'是': '是', '否': '是'}}}}
```

Calculate accuracy

```
print("C4.5 accuracy: ", simrate(test2, result2))
```

```
C4.5 accuracy:  60.00%
```

It can be seen that the accuracy of C4.5 decision tree is mediocre.

Hierarchical Clustering Experiment

Experiment Description

聚类就是对大量未知标注的数据集，按数据的内在相似性将数据集划分为多个类别，使类别内的数据相似度较大而类别间的数据相似度较小。

1. 基本要求：绘制聚类前后样本分布情况

   (1) 实现 single-linkage 层次聚类算法；

   (2) 实现 complete-linkage 层次聚类算法。

2. 中级要求：实现 average-linkage 层次聚类算法，绘制样本分布图。
3. 提高要求：对比上述三种算法，给出结论。

Experiment Preparition

Data Preparition

Firstly, set the randon seed = 42 to make sure the random numbers are the same in every experiment, so that the result can be reproduced. Then use mean1, mean2, mean3 to define three mean value of three three-dimentional normal distribution: [1, 1, 1], [4, 4, 4]and [8, 1, 1] respectively. Finally difine a 3*3 covariance matrix conv1[[1, 0, 0], [0, 1, 0], [0, 0, 1]].

```
mean1, mean2, mean3 = [1, 1, 1], [4, 4, 4], [8, 1, 1]
cov1 = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

data1 = np.random.multivariate_normal(mean1, cov1, 667)
data2 = np.random.multivariate_normal(mean2, cov1, 667)
data3 = np.random.multivariate_normal(mean3, cov1, 666)
X = np.vstack((data1, data2, data3))
labels = np.concatenate((np.full(667, 1), np.full(667, 2), np.full(666, 3)))

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=labels, cmap='rainbow', s=5)
plt.show()
```

Define Class Linkage

```
class Linkage:
    def __init__(self, n_clusters=2, linkage_name= 'single_linkage'):
        self.n_clusters = n_clusters
        self.labels_ = None
        self.linkage_matrix_ = None
        self.linkage_name_ = linkage_name
```

Implement single-linkage, complete-linkage, average-linkage hierarchical clustering.

- Single Linkage: calculate the minimum distance between samples in cluster A and samples in cluster B.

$$D_{pq} = min\{d_{ij} \mid x_i \in G_p, x_j \in G_q\}$$

```
    def calculate_single_linkage_min_distance(c1, c2, clusters, distances):
        min_dist = np.inf
        to_merge = (None, None)
        for p1 in clusters[c1]:
            for p2 in clusters[c2]:
                dist = distances[p1, p2]
                if dist < min_dist:
                    min_dist = dist
                    to_merge = (c1, c2)
        return min_dist, to_merge
```

- Complete Linkage: calculate the maximum distance between samples in cluster A and samples in cluster B.

$$D_{pq} = max\{d_{ij} \mid x_i \in G_p, x_j \in G_q\}$$

```
def calculate_complete_linkage_max_distance(c1, c2, clusters, distances):
        max_dist = -np.inf
        to_merge = (None, None)
        for p1 in clusters[c1]:
            for p2 in clusters[c2]:
                dist = distances[p1, p2]
                if dist > max_dist:
                    max_dist = dist
                    to_merge = (c1, c2)
        return max_dist, to_merge
```

- Average Linkage: calculate the average distance of arbitrary samples in cluster A and samples in cluster B.

$$D_{pq} = \frac{1}{n_p n_q} \sum_{x_i \in G_p} \sum_{x_j \in G_q} d_{ij}$$

```
def calculate_complete_linkage_max_distance(c1, c2, clusters, distances):
        max_dist = -np.inf
        to_merge = (None, None)
        for p1 in clusters[c1]:
            for p2 in clusters[c2]:
                dist = distances[p1, p2]
                if dist > max_dist:
                    max_dist = dist
                    to_merge = (c1, c2)
        return max_dist, to_merge
```

Clustering

1. Calculate the distance matrix of all samples: Calculating the Euclidean distance of every couple of samples to calculate the distance matrix of all samples.
2. Find the most close clusters: traverse to find the most close clusters, use branch structure to call single-linkage, complete-linkage and average-linkage.
3. Merge the nearest two clusters: merge the second cluster into the first cluster and delete the second cluster.
4. Assign labels to samples: assign labels to facilitate visualization

```
def fit(self, X):
        n_samples = X.shape[0]

        # Calculate the distance matrix of all samples
        distances = np.sqrt(np.sum((X[:, np.newaxis, :] - X[np.newaxis, :, :]) ** 2,
axis=2))
        clusters = {i: [i] for i in range(n_samples)}
        linkage_matrix = []

        while len(clusters) > self.n_clusters:
            min_dist = np.inf
            to_merge = (None, None)

            # Find the most close clusters
            for i in range(len(clusters)):
                for j in range(i + 1, len(clusters)):
                    c1, c2 = list(clusters.keys())[i], list(clusters.keys())[j]
                    if self.linkage_name_ == 'single_linkage':
                        dist, merge_pair = self.calculate_single_linkage_min_distance(c1,
c2, clusters, distances)
                    elif self.linkage_name_ == 'complete_linkage':
                        dist, merge_pair =
self.calculate_complete_linkage_max_distance(c1, c2, clusters, distances)
                    else:
                        dist, merge_pair = self.calculate_average_linkage_distance(c1, c2,
clusters, distances)

                    if dist < min_dist:
```

```
                        min_dist = dist
                        to_merge = merge_pair

            # Merge the nearest two clusters
            c1, c2 = to_merge
            clusters[c1].extend(clusters[c2])
            del clusters[c2]

            # print merge process
            linkage_matrix.append([c1, c2, min_dist, len(clusters[c1])])
            print(linkage_matrix[-1])

        self.linkage_matrix_ = np.array(linkage_matrix)

        # Assign labels to samples
        self.labels_ = np.zeros(n_samples)
        for cluster_id, points in enumerate(clusters.values()):
            for point in points:
                self.labels_[point] = cluster_id

        return self
```

Implement classes and visualize results

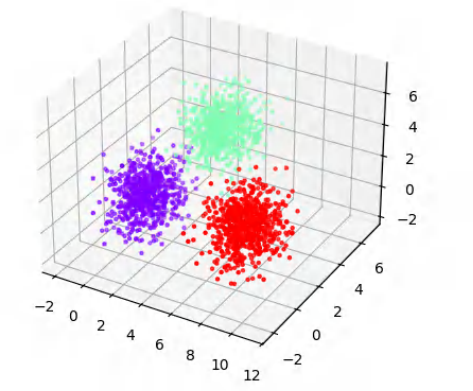
Implement single-linkage, complete-linkage and average-linkage respectively. Plot the results as scatter plots.

```
linkage_models = {
    'single_linkage': Linkage(n_clusters=3, linkage_name='single_linkage'),
    'complete_linkage': Linkage(n_clusters=3, linkage_name='complete_linkage'),
    'averaged_linkage': Linkage(n_clusters=3, linkage_name='averaged_linkage')
}

for linkage_name, model in linkage_models.items():
    model.fit(X)
    labels = model.labels_
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=labels, cmap='rainbow', s=5)
    plt.title(f"{linkage_name.capitalize()} Clustering")
    plt.show()
```

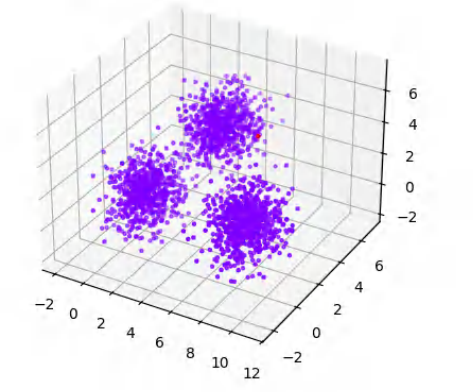
Experiment results and analysis

The experiment results are plotted below:

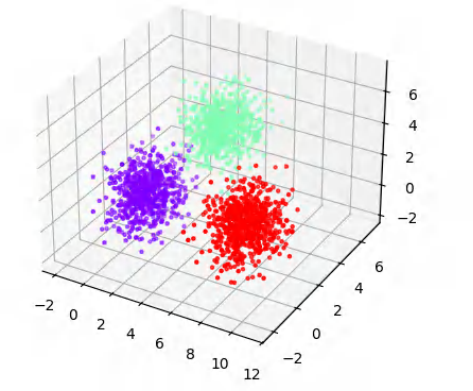
1. Original clusters and single-linkage clusters

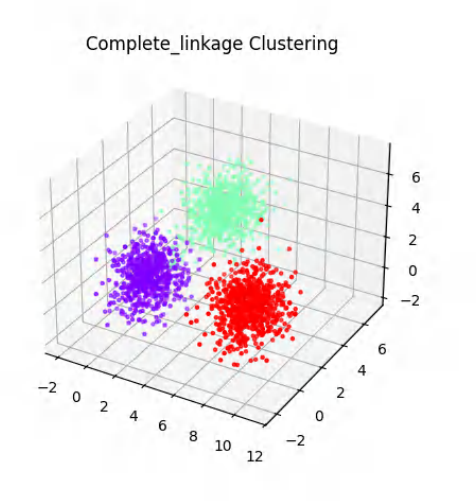


Single_linkage Clustering

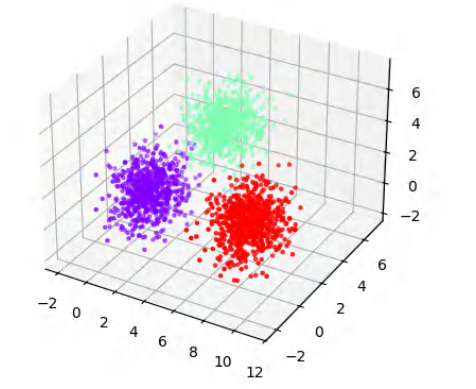


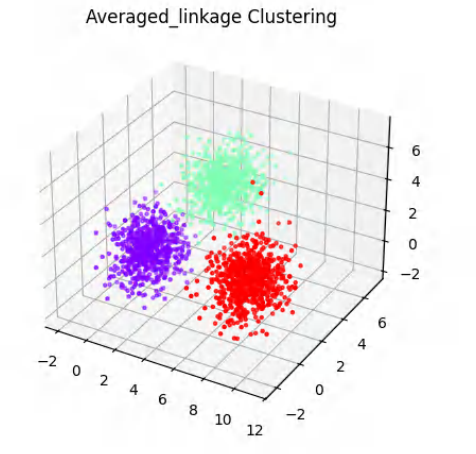
2. Original clusters and complete-linkage clusters





3. Original clusters and average-linkage clusters





As can be seen from the experimental results, the accuracy of single-linkage clustering is low -- about 2/3 samples are incorrectly classified into the red cluster. The accuracy of complete-linkage and average-linkage is higher. In complete-linkage clustering, some green samples are falsely classified into purple cluster, and a few red samples classified into green cluster. The mistakenly classified samples are few in average-linkage clustering. By analyzing the three clustering algorithm, it can be concluded that:

- Single-linkage: This method calculates the distance of the most close samples in two clusters. In other words, the distance of 2 clusters is determined by the most close samples. The method can easily lead to clusters to 'absorb' each other and have a bad classification result.

- Complete-linkage: This method tend to generate compact spherical clusters. However, during the process, two clusters can be difficult to combine because the long distance of the extreme values. Thus, the peripheral samples of the two different clusters can be poorly classified.
- Average-linkage: This method calculates the average distance of all samples in two clusters and has a balanced result.

Cat and Dog Classification with VGG16 and Resnet50

Experiment Description

使用`Kaggle`猫狗分类的原始数据集，实现模型最终的准确率达到75%及以上。

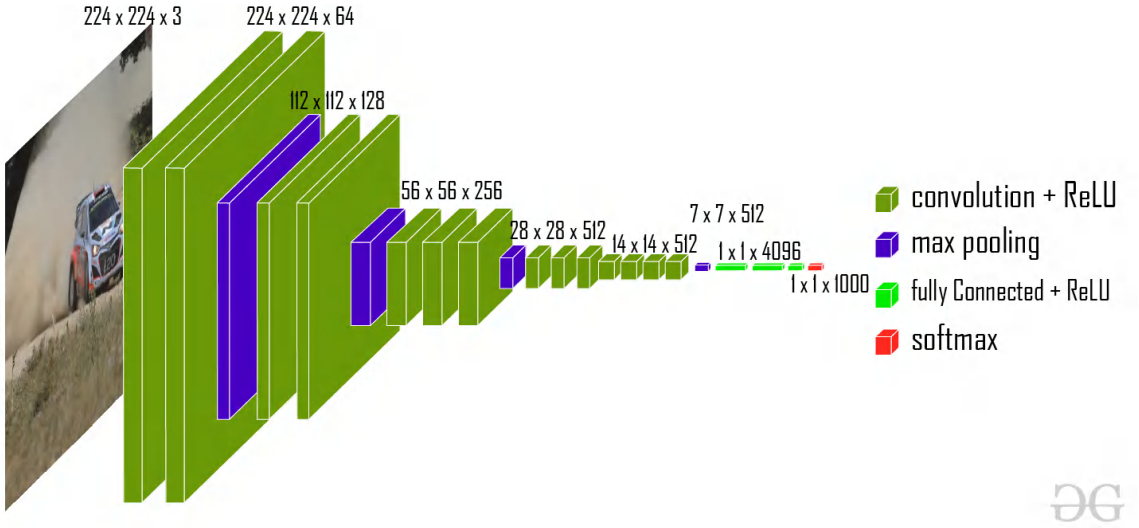
- 为了进一步掌握使用深度学习框架进行图像分类任务的具体流程如：读取数据、构造网络、训练和测试模型
- 掌握经典卷积神经网络`VGG16`、`ResNet50`的基本结构

Data Download

https://www.microsoft.com/en-us/download/details.aspx?id=54765

Experiment Principle

VGG16 stacks several small convolutional kernel and enlarges the receptive field, and achieves the same effect of using a large convolutional kernel, which can recognize bigger features. The structure of the network is very well organized, using repetitive blocks. VGG has a large amount of parameters (can reach 100 million), and takes a long training time.



ResNet mitigates the problem of slow convergence of the mode. It adds side paths in the module design, and accelerates model convergence.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Experiment Process I: VGG16

Preprocess Data

- File format preprocessing: While checking the file format, it is found that although the file extensions are all .jpg, the file actually includes formats cannot be processes like .bmp and .gif. Only .jpg,.jpeg and .png are acceptable in the network, so a seperate script is implemented to delete the files in other files from the dataset(a few dozens in total), making the model to run properly.

```
  # test.py
  import os
  from pathlib import Path
  from PIL import Image

  # dataset path
  data_dir = Path('data')

  # allowed format
  valid_formats = {'JPEG', 'PNG', 'JPG'}

  # iterate and check format
  for img_path in data_dir.glob("**/*"):
          with Image.open(img_path) as img:
              actual_format = img.format
              if actual_format not in valid_formats:
                  print(f"Deleting {img_path}, format is
{actual_format}")
                  os.remove(img_path)
```

- Data preprocessing: Use `transforms.Resize((224, 224))` to resize the pictures. Use `transforms.ToTensor` to change pictures into processable tensor, with the value between 0 and 1.

```
  import torch
  import torch.nn as nn
  import torch.optim as optim
  from torch.utils.data import DataLoader
  from torchvision import datasets, transforms
  from torch import device
  from PIL import Image

  # preprocess: size, tensor, normalization
  transform = transforms.Compose([
      transforms.Resize((224, 224)),
      transforms.ToTensor(),
      transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
```

Dataset Procession

Load dataset as train_data from `data` folder, use transform defined above to preprocess. Define iterator train_loader, randomly load 32 data per batch.

```
train_data = datasets.ImageFolder('data', transform=transform)
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
```

Use mps acceleration.

```
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')
```

Define VGG16

Use 3*3 convolutional kernels. The first two layer groups are convolution-convolution-pooling, the last three layer groups are convolution-convolution-convolution-pooling, and three full connection layers, 16 layers in total.

```
# VGG16.py
import torch.nn as nn

# VGG16
```

```
class VGG16(nn.Module):
    def __init__(self, num_classes=2):
        super(VGG16, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),

            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
```

```
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, num_classes),  # output 2 classes
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)  # flatten
        x = self.classifier(x)
        return x
```

Define ResNet50

```
import torch
import torch.nn as nn
from torch.autograd import Variable

# check MPS
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

class convolutional_block(nn.Module):  # convolutional_block
    def __init__(self, cn_input, cn_middle, cn_output, s=2):
        super(convolutional_block, self).__init__()
        self.step1 = nn.Sequential(nn.Conv2d(cn_input, cn_middle, (1, 1), (s,
s), padding=0, bias=False),
                                   nn.BatchNorm2d(cn_middle, affine=False),
nn.ReLU(inplace=True),
                                   nn.Conv2d(cn_middle, cn_middle, (3, 3),
(1, 1), padding=(1, 1), bias=False),
                                   nn.BatchNorm2d(cn_middle, affine=False),
nn.ReLU(inplace=True),
                                   nn.Conv2d(cn_middle, cn_output, (1, 1),
(1, 1), padding=0, bias=False),
                                   nn.BatchNorm2d(cn_output, affine=False))
        self.step2 = nn.Sequential(nn.Conv2d(cn_input, cn_output, (1, 1), (s,
s), padding=0, bias=False),
                                   nn.BatchNorm2d(cn_output, affine=False))
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x_tmp = x
        x = self.step1(x)
        x_tmp = self.step2(x_tmp)
        x = x + x_tmp
```

```
        x = self.relu(x)
        return x


class identity_block(nn.Module):  # identity_block
    def __init__(self, cn, cn_middle):
        super(identity_block, self).__init__()
        self.step = nn.Sequential(nn.Conv2d(cn, cn_middle, (1, 1), (1, 1),
padding=0, bias=False),
                                  nn.BatchNorm2d(cn_middle, affine=False),
nn.ReLU(inplace=True),
                                  nn.Conv2d(cn_middle, cn_middle, (3, 3), (1,
1), padding=1, bias=False),
                                  nn.BatchNorm2d(cn_middle, affine=False),
nn.ReLU(inplace=True),
                                  nn.Conv2d(cn_middle, cn, (1, 1), (1, 1),
padding=0, bias=False),
                                  nn.BatchNorm2d(cn, affine=False))
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x_tmp = x
        x = self.step(x)
        x = x + x_tmp
        x = self.relu(x)
        return x


class Resnet(nn.Module):  # main model
    def __init__(self, c_block, i_block):
        super(Resnet, self).__init__()
        self.conv = nn.Sequential(nn.Conv2d(3, 64, (7, 7), (2, 2), padding=
(3, 3), bias=False),
                                  nn.BatchNorm2d(64, affine=False),
nn.ReLU(inplace=True), nn.MaxPool2d((3, 3), 2, 1))
        self.layer1 = c_block(64, 64, 256, 1)
        self.layer2 = i_block(256, 64)
        self.layer3 = c_block(256, 128, 512)
        self.layer4 = i_block(512, 128)
        self.layer5 = c_block(512, 256, 1024)
        self.layer6 = i_block(1024, 256)
        self.layer7 = c_block(1024, 512, 2048)
        self.layer8 = i_block(2048, 512)
        self.out = nn.Linear(2048, 2, bias=False)
        self.avgpool = nn.AvgPool2d(7, 7)
```

```
    def forward(self, input):
        x = self.conv(input)
        x = self.layer1(x)
        for i in range(2):
            x = self.layer2(x)
        x = self.layer3(x)
        for i in range(3):
            x = self.layer4(x)
        x = self.layer5(x)
        for i in range(5):
            x = self.layer6(x)
        x = self.layer7(x)
        for i in range(2):
            x = self.layer8(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output

# implement and move to mps
net = Resnet(convolutional_block, identity_block).to('mps')
```

Define Hyperparameters

This is a binary classification problem, num_classes=2, move model to mps to train. Use cross-entropy for classification problem. Use Adam optimizer, with a learning rate of 0.0001, train for 10 epochs.

```
# implement model
# model = VGG16(num_classes=2)
# model = models.vgg16(pretrained=True)
model = models.resnet50(pretrained=True)
# model = Resnet(convolutional_block, identity_block)

model = model.to(device)

# loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

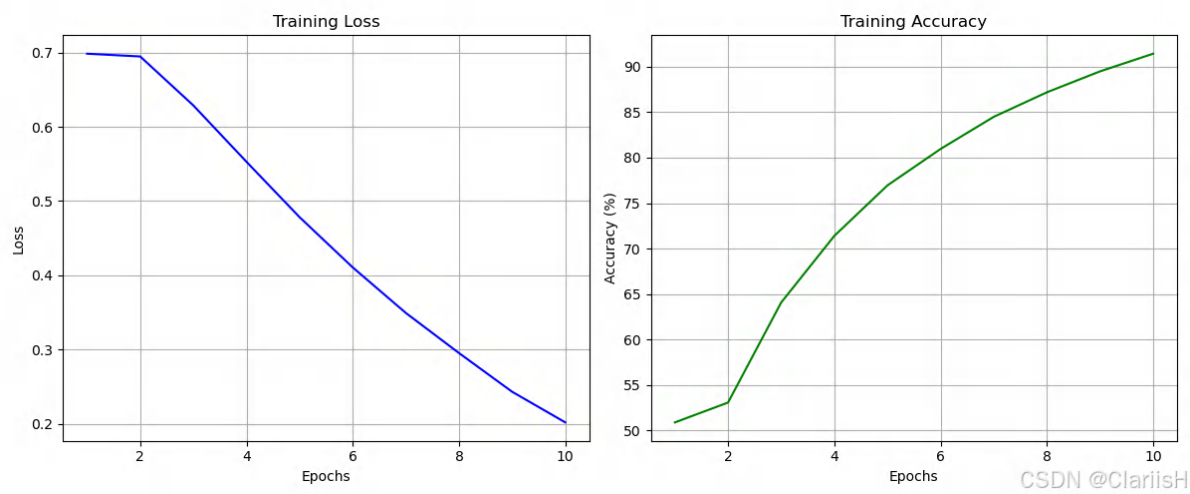
Training

Train the model for 10 epochs, save the model, and plot the result.

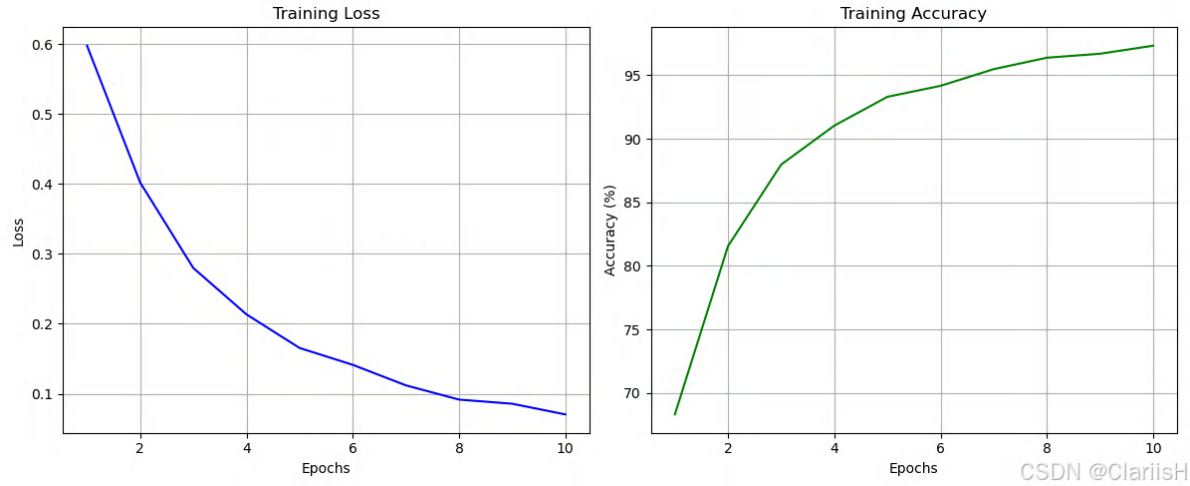
```
for epoch in range(num_epochs):
    model.train()  # train mode
    running_loss = 0.0
    correct = 0
    total = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # empty gradients, prevent accumulatin
        optimizer.zero_grad()
        outputs = model(inputs) # forward
        loss = criterion(outputs, labels)
        loss.backward() # backward
        optimizer.step()

        running_loss += loss.item()

        # accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss:
{running_loss/len(train_loader):.4f}, Accuracy: {100 * correct /
total:.2f}%")
```

Analysis

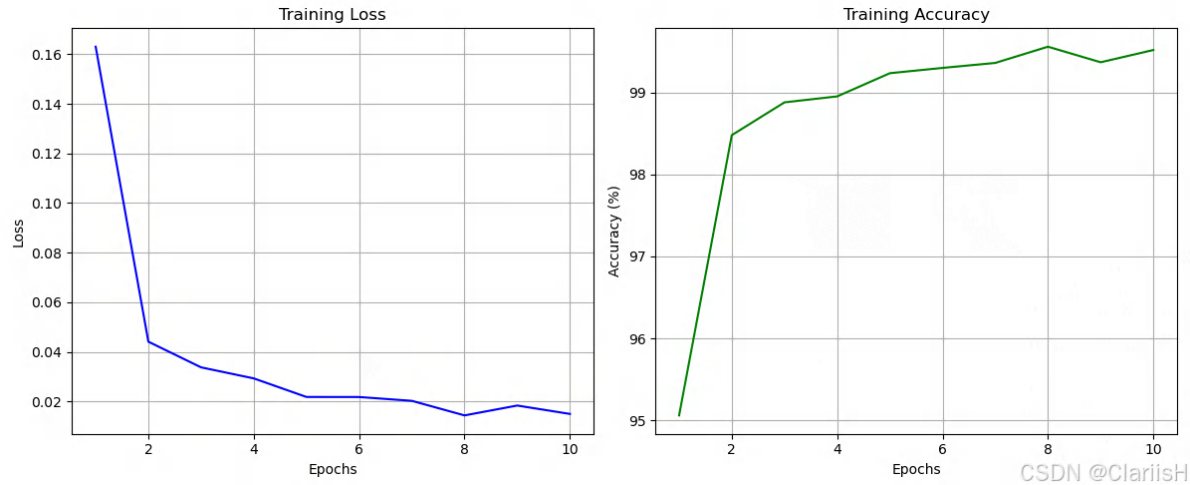
It can be seen from the output that the accuracy of VGG16 rised rapidly after two epochs of training. From the sixth epoch the accuracy increase slows down, and reach 93.78% in the tenth epoch. The performance of the model is good.



Resnet has a high accuracy from the beginning(70% approximately). Then it reached over 95% accuracy in the 10th epoch.



For pre-trained models, VGG16 has the accuracy of 95% from the beginning. After training for four epochs, the accuracy reached 99%.



The pre-trained Resnet50 model reached the accuracy of 97% from the beginning, and the accuracy raised to 99% after two epochs.

It can be seen from the experiment above that, ResNet50 has faster convergence speed and higher accuracy than VGG16, including the model built from scratch and the pre-trained model. Both of the pre-trained models have good performance.

Naive Bayes Classifier

Experiment Description

数据集

Most Popular Data Set中的wine数据集（对意大利同一地区声场的三种不同品种的酒做大量分析所得出的数据）

基本要求

a)采用分层采样的方式将数据集划分为训练集和测试集。

b)给定编写一个朴素贝叶斯分类器，对测试集进行预测，计算分类准确率。

中级要求

使用测试集评估模型，得到混淆矩阵，精度，召回率，F值。

高级要求

在中级要求的基础上画出三类数据的ROC曲线，并求出AUC值。

Experiment Process

1. Import libraries and dataset

```
import numpy as np
import math
import random
import csv
import operator

with open('wine.data') as csvfile:
     reader = csv.reader(csvfile)
     dataset = [row for row in reader]
```

2. Preprocess and spilt dataset as 7:3 into train dataset and test dataset. Data is stored in strings, so transform it into float. Use stratified sampling on train dataset. The numbers of three datasets are 42, 50, 33.

```
x_train = []
y_train = []
x_test = []
y_test = []
seed = []
train = random.sample(dataset[0: 59], 42)
train = train + random.sample(dataset[59: 59 + 71], 50)
train = train + random.sample(dataset[59 + 71: -1], 33)
test = [i for i in dataset if i not in train]

def to_float(dataset):
```

```
    y = []
    for i in dataset:
        for m in range(len(i)):
            i[m] = float(i[m])
        y.append(int(i[0]))
        i.pop(0)
    return dataset, y

x_train, y_train = to_float(train)
x_test, y_test = to_float(test)
```

3. Calculate probability density function.

   $P(c|x)$: posterior probability（given sample x，the probability of belonging to class c)

   $P(x|c)$: Assume the sample is in class c, the probability of observing sample x.

   $P(c)$: The prior probability of sample in class c. In reallife applications, the prior probability is usually unknown. It can only be asumed based on background knowledge, experiences, and training data. This is one of the difficulties of Bayes Classifier. Under this specific circumstance, $P(c)$ is assumed as normal distribution, with a probability density function of $\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-u)^2}{2\sigma^2}}$.

```
def Bayes(data, p, avg, var):
    result = p
    for i in range(len(data)):
        result *=  1 / (math.sqrt(2 * math.pi * var[i])) * math.exp(-((data[i] -
avg[i])**2) / (2 * var[i]))
    return result
```

4. Classification

   By calculating posterior probability, classify test dataset, and calculate the accuracy.

```
def classifier(x_train, x_test):
    result = []
    x_train = np.array(x_train)
    avg1 = x_train[:42].mean(axis = 0)
    var1 = x_train[:42].var(axis = 0)
    avg2 = x_train[42 : 42 + 50].mean(axis = 0)
    var2 = x_train[42 : 42 + 50].var(axis = 0)
    avg3 = x_train[42 + 50 : ].mean(axis = 0)
    var3 = x_train[42 + 50 : ].var(axis = 0)
    for i in range(len(x_test)):
        temp = 1
        max = Bayes(x_test[i], 59 / (59 + 71 + 48), avg1, var1)
        if Bayes(x_test[i], 71 / (59 + 71 + 48), avg2, var2) > max:
            temp = 2
            max = Bayes(x_test[i], 71 / (59 + 71 + 48), avg2, var2)
        if Bayes(x_test[i], 48 / (59 + 71 + 48), avg3, var3) > max:
            temp = 3
        result.append(temp)
    return result
```

```
def simrate(ls1, ls2):
    num = 0
    l = len(ls1)
    for i in range(l):
        if ls1[i] == ls2[i]:
            num += 1
    return format(num / l, '.2%')

predict = classifier(x_train, x_test)

print("分类的准确率是", simrate(predict, y_test))
```

5. Generate confusion matrix

```
    def confuse_maxtria(predict, fact):
        ls = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
        for i in range(len(predict)):
            ls[fact[i] - 1][predict[i] - 1] += 1
        return ls

    print("混淆矩阵是:", confuse_maxtria(predict, y_test))
```

6. Calculate precision, recall and F-measure
   - True Positive: Predict positive values as positive class
   - True Negative: Predict negative values as negative class
   - False Positive: Predict negative values as positive class (Type I error)
   - False Negative: Predict positive values as negative class (Type II error)
   - Precision: $p = \frac{TP}{TP+FP}$
   - Recall: $p = \frac{TP}{TP+FN}$
   - F-measure: $F - Measure = \frac{(a^2+1)P*R}{P+R}$，本次实验中$a$值取为1，a>1 recall weight more; a<1 precision weight more

```
    def get_feature(confuse_maxtria):
        for index in range(len(confuse_maxtria)):
            truth = confuse_maxtria[index][index]
            total = 0
            total2 = 0
            for i in range(len(confuse_maxtria)):
                total += confuse_maxtria[index][i]
            for i in range(len(confuse_maxtria)):
                total2 += confuse_maxtria[i][index]
            precision = truth / total
            recall = truth / total2
            f_rate = 2 * precision * recall / (precision + recall)
            print("类别", index + 1, "的精度为", precision, ", 召回率为", recall, ", F值为", f_rate)
```

```
    get_feature(confuse_maxtria(predict, y_test))
```

Experiment Result

```
分类的准确率是 96.23%
混淆矩阵是: [[16, 1, 0], [0, 20, 1], [0, 0, 15]]
类别 1 的精度为 0.9411764705882353 , 召回率为 1.0 , F值为 0.9696969696969697
类别 2 的精度为 0.9523809523809523 , 召回率为 0.9523809523809523 , F值为 0.9523809523809523
类别 3 的精度为 1.0 , 召回率为 0.9375 , F值为 0.967741935483871
```

It can be seen that this experiment have a high accuracy, and the evaluation indexes(precision, recall and F-measure) are good. The classification outcome is ideal.

Parametric and Non-Parametric Estimation

Experiment Description

基本要求 (3')

在两个数据集合上应用“最大后验概率规则” 进行分类实验，计算分类错误率，分析实验结果。

中级要求 (2')
在两个数据集合上使用高斯核函数估计方法，应用“似然率测试规则”分类，在 [0.1, 0.5, 1, 1.5, 2] 范围内交叉验证找到最优 h 值，分析实验结果。

Data Generation

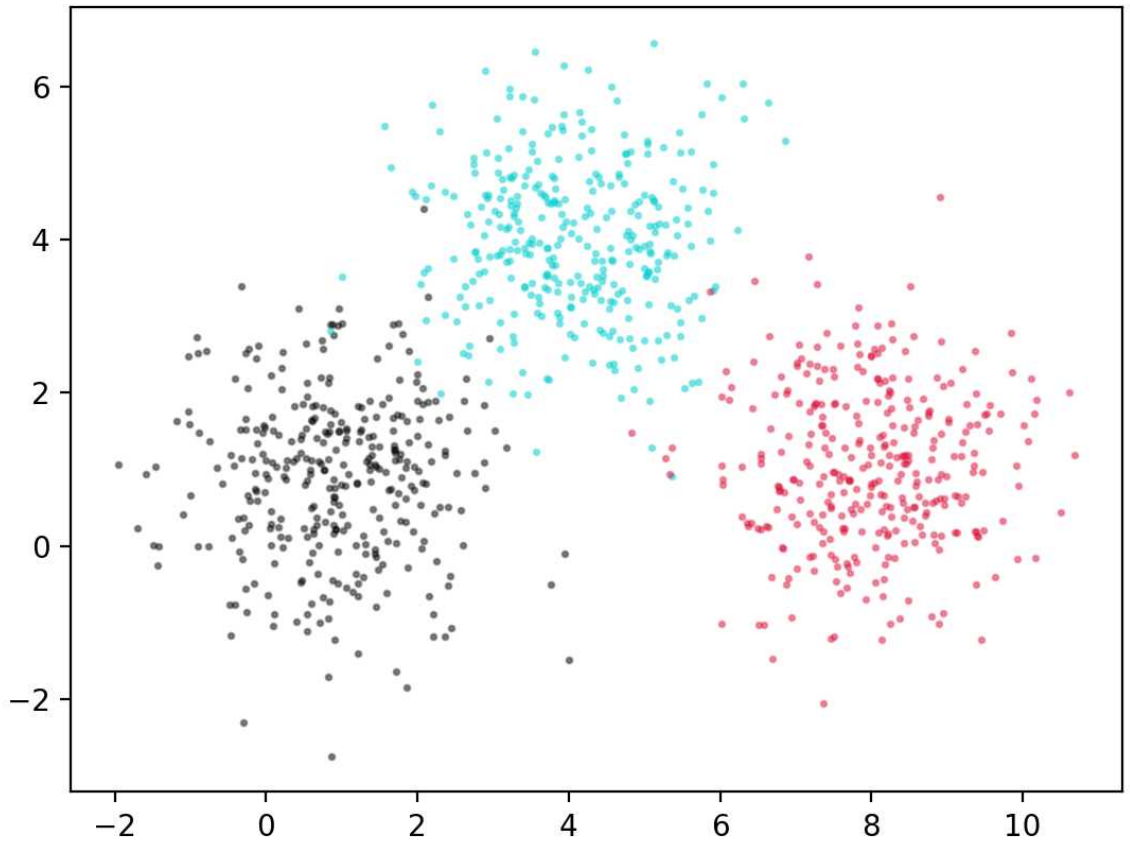
Data: generate two dataset X_1, X_2, including N=1000 two-dimentional random vectors respectively. The random vectors are from three distributions, which has mean value $m1 = [1, 1]^T, m2 = [4, 4]^T$, $m3 = [8, 1]^T$ and covariance matrix S1 = S2 = S3 = 2(I is 2*2 indentity matrix).

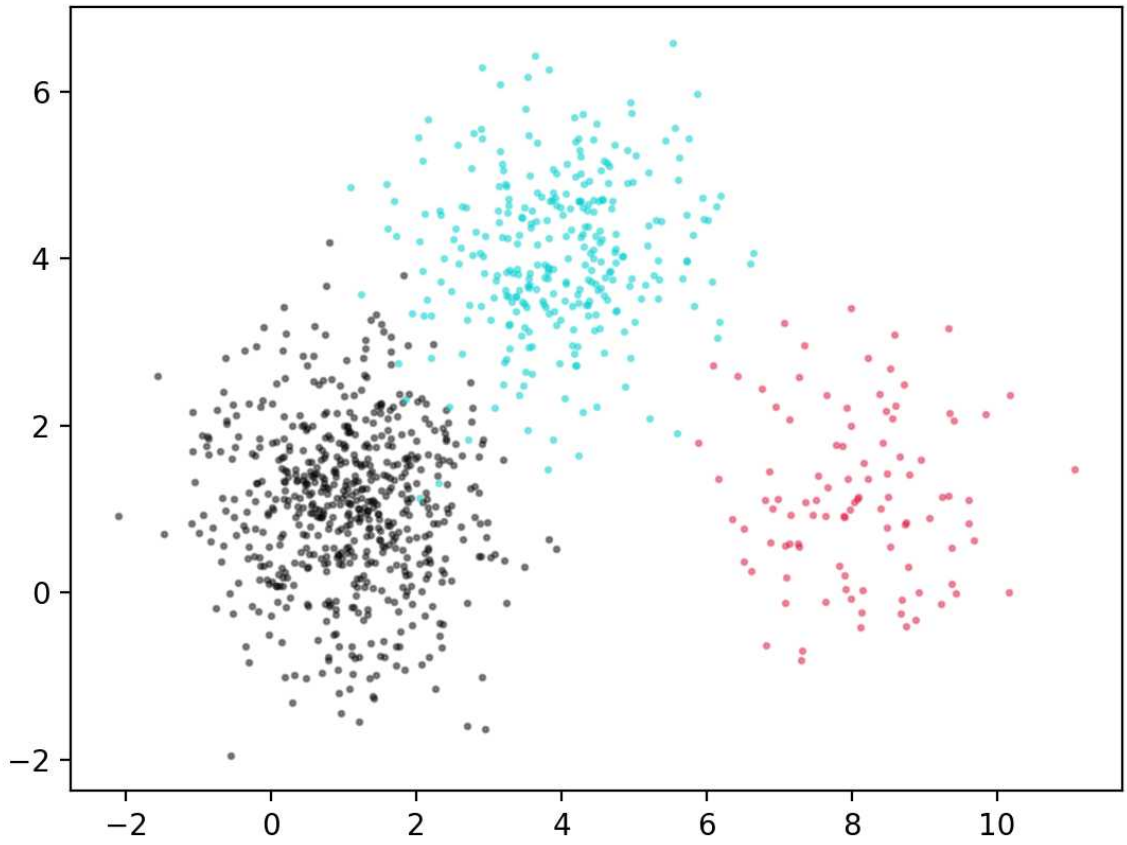
Use random.multivariate_normal from numpy library to generate data.

```
y as np
import matplotlib.pyplot as plt
import math
def f(m, cnt):
    x = []
    y = []
    for i in range(cnt):
        x.append(m[i][0])
        y.append(m[i][1])
    return x, y

def random_x(cnt1, cnt2, cnt3, name):
    cov = [[1, 0], [0, 1]]
    a1 = np.random.multivariate_normal((1, 1), cov, cnt1)
    a2 = np.random.multivariate_normal((4, 4), cov, cnt2)
    a3 = np.random.multivariate_normal((8, 1), cov, cnt3)
    colors0 = '#000000'
    colors1 = '#00CED1' # color
    colors2 = '#DC143C'
    area = np.pi   # area
    x, y = f(a1, cnt1)
    plt.scatter(x, y, s=area, c=colors0, alpha=0.4)
    x, y = f(a2, cnt2)
    plt.scatter(x, y, s=area, c=colors1, alpha=0.4)
    x, y = f(a3, cnt3)
    plt.scatter(x, y, s=area, c=colors2, alpha=0.4)
    ls = []
    result = []
    for i in range(cnt1):
        ls.append(a1[i])
```

```
            result.append(1)
        for i in range(cnt2):
            ls.append(a2[i])
            result.append(2)
        for i in range(cnt3):
            ls.append(a3[i])
            result.append(3)
        plt.figure(num = name)
        return ls, result

x1, c1 = random_x(333, 333, 334)
x2, c2 = random_x(600, 300, 100)
```

The scatter plot of generated data:





Use Maximum Posterior Probability for Classification

1. Calculate according to probability density function $\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(\theta-u)^2}{2\sigma^2}}$.

```
def Normal_distribution(data, m):
    m = np.array(m)
    cov = np.array([[1, 0], [0, 1]])
    return 1 / math.sqrt((2 * np.pi) ** 2 * 1) * math.exp(-0.5 * np.dot((data - m).T,
(data - m)))
```

2. Calculate posterior probability respectively and classify the data point into the cluster with maximum posterior probability.

```
    def Classification(data, mean1, mean2, mean3):
        result = []
        for d in data:
            t1 = Normal_distribution(d, mean1)
            t2 = Normal_distribution(d, mean2)
            t3 = Normal_distribution(d, mean3)
            i = 1
            max = t1
            if t2 > max:
                max = t2
```

```
            i = 2
        if t3 > max:
            max = t3
            i = 3
        result.append(i)
    return result
```

3. Calculate accuracy and visualize with scatter plot.

```
def simrate(ls1, ls2):
    num = 0
    l = len(ls1)
    for i in range(l):
        if ls1[i] != ls2[i]:
            num += 1
    return format(num / l, '.2%')

def show_result(data, result, name):
    colors0 = '#000000'
    colors1 = '#00CED1' # color
    colors2 = '#DC143C'
    area = np.pi
    x1 = []
    x2 = []
    x3 = []
    y1 = []
    y2 = []
    y3 = []
    for i in range(len(data)):
        if result[i] == 1:
            x1.append(data[i][0])
            y1.append(data[i][1])
        elif result[i] == 2:
            x2.append(data[i][0])
            y2.append(data[i][1])
        elif result[i] == 3:
            x3.append(data[i][0])
            y3.append(data[i][1])
    plt.scatter(x1, y1, s=area, c=colors0, alpha=0.4)
    plt.scatter(x2, y2, s=area, c=colors1, alpha=0.4)
    plt.scatter(x3, y3, s=area, c=colors2, alpha=0.4)
    plt.figure(num = name)

result1 = Classification(x1, (1, 1), (4, 4), (8, 1))
result2 = Classification(x2, (1, 1), (4, 4), (8, 1))

Error_rate1 = simrate(result1, c1)
Error_rate2 = simrate(result2, c2)

show_result(x1, result1)
show_result(x2, result2)
```

```
    plt.show()

    print("\nX_1数据集使用最大后验概率规则进行分类的错误率是", Error_rate1)
    print("X_2数据集使用最大后验概率规则进行分类的错误率是", Error_rate2)
```

Use Kernel Density Estimation (KDE) for Classification

1. Calculate according to probability density function $\frac{1}{N}\sum_{n=1}^{N}\frac{1}{\sqrt{2\pi h^2}}exp\{-\frac{||x-x_n||^2}{2h^2}\}$

```
    def guss(x, data, h):
        n = len(data)
        result = 0
        for i in range(n):
            result += math.exp(- (math.sqrt((x[0] - data[i][0]) ** 2 + (x[1] - data[i]
[1]) ** 2)) / (2 * h * h))
        result = result / (n * math.sqrt(2 * np.pi * h * h))
        return result
```

2. Calculate likelihood respectively and classify the data point into the cluster with maximum likelyhood.

```
    def Classification_2(data, m1, m2, m3, h, r):
        result = []
        data1 = data[0 : m1]
        data2 = data[m1 : m1+m2]
        data3 = data[m1+m2 : m1+m2+m3]
        for d in data:
            t1 = guss(d, data1, h)
            t2 = guss(d, data2, h)
            t3 = guss(d, data3, h)
            i = 1
            max = t1
            if t2 > max:
                max = t2
                i = 2
            if t3 > max:
                max = t3
                i = 3
            result.append(i)
        return simrate(result, r)
```

3. Classification and visualization.

```
r1_1 = Classification_2(x1, 334, 333, 333, 0.1, c1)
r1_5 = Classification_2(x1, 334, 333, 333, 0.5, c1)
r1_10 = Classification_2(x1, 334, 333, 333, 1, c1)
r1_15 = Classification_2(x1, 334, 333, 333, 1.5, c1)
r1_20 = Classification_2(x1, 334, 333, 333, 2, c1)
```

```
r2_1 = Classification_2(x2, 600, 300, 100, 0.1, c2)
r2_5 = Classification_2(x2, 600, 300, 100, 0.5, c2)
r2_10 = Classification_2(x2, 600, 300, 100, 1, c2)
r2_15 = Classification_2(x2, 600, 300, 100, 1.5, c2)
r2_20 = Classification_2(x2, 600, 300, 100, 2, c2)
print("X_1数据集利用似然率测试规则在h = 0.1情况下分类错误率是 ", r1_1)
print("X_1数据集利用似然率测试规则在h = 0.5情况下分类错误率是 ", r1_5)
print("X_1数据集利用似然率测试规则在h = 1.0情况下分类错误率是 ", r1_10)
print("X_1数据集利用似然率测试规则在h = 1.5情况下分类错误率是 ", r1_15)
print("X_1数据集利用似然率测试规则在h = 2.0情况下分类错误率是 ", r1_20)

print("X_2数据集利用似然率测试规则在h = 0.1情况下分类错误率是 ", r2_1)
print("X_2数据集利用似然率测试规则在h = 0.5情况下分类错误率是 ", r2_5)
print("X_2数据集利用似然率测试规则在h = 1.0情况下分类错误率是 ", r2_10)
print("X_2数据集利用似然率测试规则在h = 1.5情况下分类错误率是 ", r2_15)
print("X_2数据集利用似然率测试规则在h = 2.0情况下分类错误率是 ", r2_20)
```

Experiment Result

The main difference of parametric and non-parametric estimation is that whether assumpt data is from a specific distribution.

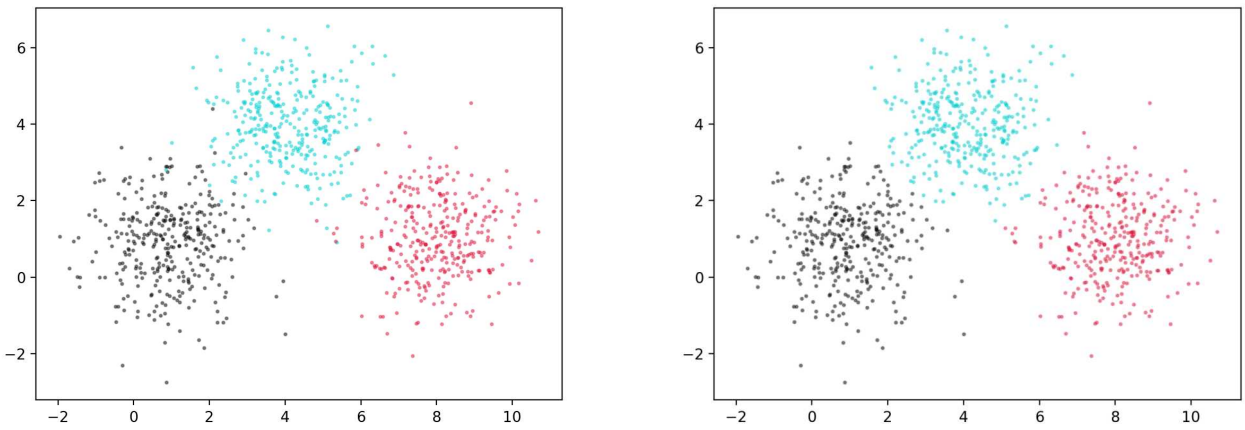
Use Maximum Posterior Probability for Classification

The accuracy is shown as below:

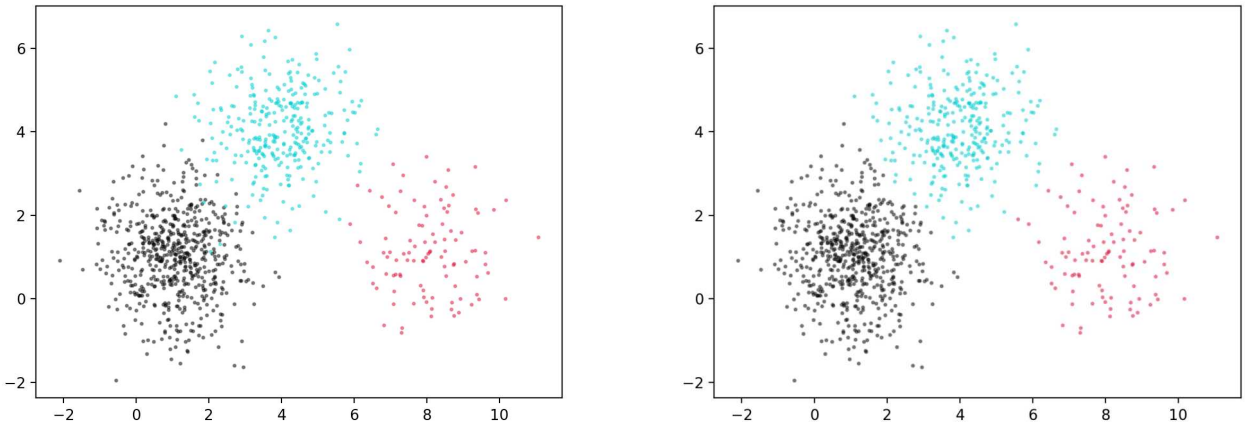
```
X_1数据集使用最大后验概率规则进行分类的错误率是 1.20%
X_2数据集使用最大后验概率规则进行分类的错误率是 2.00%
```

Use scatter plot to compare with original data:

X_1(Original data, left) compared with data classified with Maximum Posterior Probability in X_1(Right)



X_2(Original data, left) compared with data classified with Maximum Posterior Probability in X_2(Right)



In conclusion, using Maximum Posterior Probability for classification has high accuracy, and the classification errors mostly happen to peripheral samples. The clssification technique has a ideal effect and can be implemented.

Use Kernel Density Estimation (KDE) for Classification

```
X_1数据集利用似然率测试规则在h = 0.1情况下分类错误率是  0.20%
X_1数据集利用似然率测试规则在h = 0.5情况下分类错误率是  0.90%
X_1数据集利用似然率测试规则在h = 1.0情况下分类错误率是  1.10%
X_1数据集利用似然率测试规则在h = 1.5情况下分类错误率是  1.20%
X_1数据集利用似然率测试规则在h = 2.0情况下分类错误率是  1.20%
X_2数据集利用似然率测试规则在h = 0.1情况下分类错误率是  0.20%
X_2数据集利用似然率测试规则在h = 0.5情况下分类错误率是  1.90%
X_2数据集利用似然率测试规则在h = 1.0情况下分类错误率是  1.90%
X_2数据集利用似然率测试规则在h = 1.5情况下分类错误率是  2.00%
X_2数据集利用似然率测试规则在h = 2.0情况下分类错误率是  2.00%
```

It can be seen that in the KDE classification experiment, the large size of window(h) can generate an overly smooth probability desity estimation. It makes the structure of data inaccurate and vague, and overlooks data details. If h is too small, the probability desity estimation has a lot of peaks, which makes it hard to find a clear trend. The experiment shows that for the two datasets, the window size h=0.1 has the best effect, where the error rates are both 0.2%.

Regression Model

Experiment Description

实验基本要求

- 根据数据集 dataset_regression.csv ，求最小二乘解，画出回归曲线，给出训练误差。
- 将数据集 winequality-white.csv 按照4:1划分为训练集和测试集，构造线性回归模型，采用批量梯度下降或者随机梯度下降均可;输出训练集和测试集的均方误差(MSE)，画出MSE收敛曲线。

实验中级要求

尝试不同的学习率并进行MSE曲线展示，分析选择最佳的学习率。

Analysis

Least Square Fit

```
import numpy as np
import matplotlib.pyplot as plt
import csv
import operator
with open('dataset_regression.csv') as csvfile:
     reader = csv.reader(csvfile)
     dataset = [row for row in reader]
dataset.pop(0)
for i in dataset:
    for m in range(3):
        i[m] = float(i[m])
print(dataset)
n = len(dataset)
sum_xy = 0
sum_x = 0
sum_y = 0
sum_xx = 0
for i in range(n):
    sum_xy += dataset[i][1] * dataset[i][2]
    sum_x += dataset[i][1]
    sum_y += dataset[i][2]
    sum_xx += dataset[i][1] * dataset[i][1]
a1 = (sum_xy - (sum_x * sum_y) / n) / (sum_xx - n * (sum_x / n  * sum_x / n))
a0 = sum_y / n - a1 * sum_x / n
a1 = round(a1, 4)
a0 = round(a0, 1)
print("回归方程为: y=", a1 , "x+" , a0)
xt = []
yt = []
for i in dataset:
    xt.append(i[1])
    yt.append(i[2])
```

```
    loss = 0
    for i in range(n):
        loss += (yt[i] - a0 - a1 * xt[i]) ** 2
    loss = loss / (2 * n)
    print("loss的值为：", loss)
    fig = plt.figure(figsize=(4, 4))
    ax = fig.add_subplot(1, 1, 1)
    ax.scatter(xt, yt)
    x = np.arange(-3, 4)
    y = a1 * x + a0
    plt.plot(x, y)
    plt.show()
```

- Use batch gradient descent to construct the model for multi-dimentional data. Before using gradient descent, the data should be normalized first. Set learning rate at 0.5 0.3 0.1 0.01 0.001 and test.

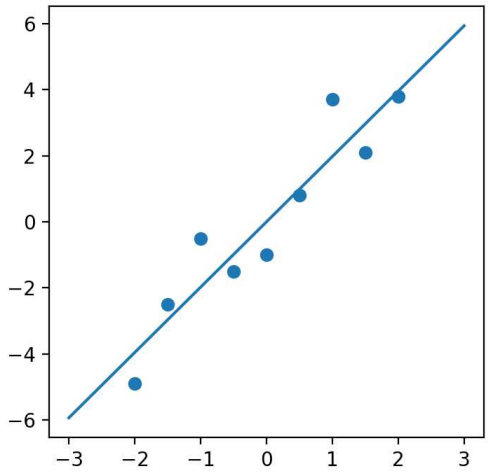
```
    import numpy as np
    import csv
    import operator
    from sklearn.model_selection import train_test_split
    import matplotlib.pyplot as plt
    with open('winequality-white.csv') as csvfile:
         reader = csv.reader(csvfile)
         dataset = [row for row in reader]
    dataset.pop(0)
    y = []
    for i in dataset:
        for m in range(len(i)):
            i[m] = float(i[m])
        y.append(i[-1])
        i.pop(-1)
        i.insert(0, 1)
    x_train, x_test, y_train, y_test = train_test_split(dataset, y, test_size = 0.2) #划分训
练集
    #归一化
    def feature_scaling(x):
        for i in range(len(x[0])):
            max = -float('inf')
            min = float('inf')
            for m in range(len(x)):
                if x[m][i] > max:
                    max = x[m][i]
                if x[m][i] < min:
                    min = x[m][i]
            for m in range(len(x)):
                if max - min != 0:
                    x[m][i] = (x[m][i] - min) / (max - min)
        return x
    x_train = feature_scaling(x_train)
    x_test = feature_scaling(x_test)
    # theta = np.random.rand(len(x_train[0]))
```

```
  # print(x_train[0])
  # print(theta)
  # print(theta * x_train)
  # print(x_train)
  # print(type(np.random.rand(len(x_train))))
  def gradient_descent(x_train, y_train, x_test, y_test, learning_rate):
      loss = []
      theta = np.random.rand(len(x_train[0]))
      x_train = np.array(x_train)
      x_test = np.array(x_test)
      for index in range(1000):
          gradients = x_train.T.dot(x_train.dot(theta) - y_train) / len(x_train)
          theta = theta - learning_rate * gradients
          MSE = ((np.dot(x_test, theta) - y_test) ** 2).sum() / len(x_test)
          loss.append(MSE)
      return theta, loss
  ls = []
  for i in range(1000):
      ls.append(i)
  x = np.array(ls)
  theta0, loss0 = gradient_descent(x_train, y_train, x_train, y_train, learning_rate =
0.5)
  theta1, loss1 = gradient_descent(x_train, y_train, x_test, y_test, learning_rate = 0.5)
  theta2, loss2 = gradient_descent(x_train, y_train, x_test, y_test, learning_rate = 0.3)
  theta3, loss3 = gradient_descent(x_train, y_train, x_test, y_test, learning_rate = 0.1)
  theta4, loss4 = gradient_descent(x_train, y_train, x_test, y_test, learning_rate = 0.01)
  theta5, loss5 = gradient_descent(x_train, y_train, x_test, y_test, learning_rate =
0.001)
  # 画散点图
  colors0 = '#000000'
  colors1 = '#00CED1' #点的颜色
  colors2 = '#DC143C'
  colors3 = '#66CDAA'
  colors4 = '#BEBEBE'
  colors5 = '#00FA9A'
  area = np.pi * 0.5**2  # 点面积
  plt.scatter(x, loss0, s=area, c=colors0, alpha=0.4, label='train')
  plt.scatter(x, loss1, s=area, c=colors1, alpha=0.4, label='learning_rate = 0.5')
  plt.scatter(x, loss2, s=area, c=colors2, alpha=0.4, label='learning_rate = 0.3')
  # plt.scatter(x, loss3, s=area, c=colors3, alpha=0.4, label='learning_rate = 0.1')
  # plt.scatter(x, loss4, s=area, c=colors4, alpha=0.4, label='learning_rate = 0.01')
  # plt.scatter(x, loss5, s=area, c=colors5, alpha=0.4, label='learning_rate = 0.001')
  plt.legend()
  plt.show()
```

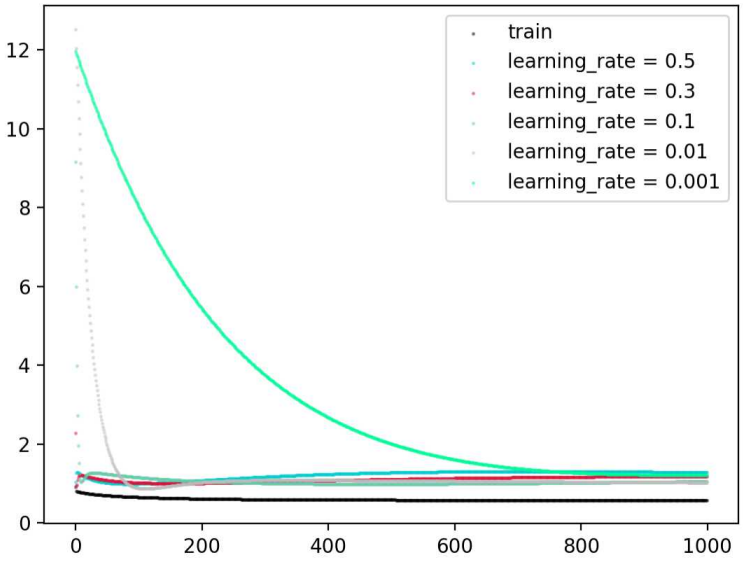
Experiment Result

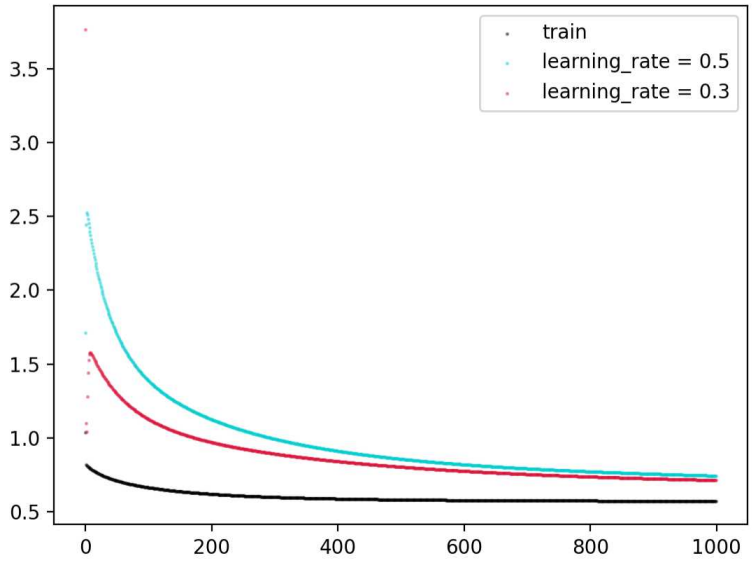
- Least Square Fit

```
[[1.0, -2.0, -4.9], [2.0, -1.5, -2.5], [3.0, -1.0, -0.5], [4.0, -0.5, -1.5], [5.0, 0.0, -1.0], [6.0, 0.5, 0.8], [7.0, 1.0, 3.7], [8.0, 1.5, 2.1], [9.0, 2.0, 3.8]]
回归方程为: y= 1.9767 x+ 0.0
loss的值为:  0.46287963055555564
```



- Gradient Descent





It can be concluded from the data that, extremely high or low learning rate can affect accuracy. Extremely high learning rate can lead to overfitting, and extremely low learning rate results in underfitting. The analysis shows that the optimal learning rate of this model is about 0.3.

Handwritten Digit Recognition Based on K Nearest Neighbourhood

Experiment Description

数据集：semeion_train.csv, semeion_test.csv

实验基本要求

编程实现kNN算法；给出在不同k值（1，3，5）情况下，kNN算法对手写数字的识别精度。

实验中级要求

与 Python 机器学习包中的kNN分类结果进行对比。

Experiment Process

Import Libraries

```
import numpy as np
import csv
import operator
```

Read Data

Read train data and test data respectively and store in lists.

```
train = []
train_result = []
test = []
test_right = []
# import data
with open('semeion_train.csv') as csvfile:
    reader = csv.reader(csvfile)
    rows= [row for row in reader]
for i in rows:
    ls = []
    temp = i[0].split()
    num = 0
    for m in temp[:-10]:
        m = float(m)
        ls.append(m)
    for m in temp[-10:]:
        m = int(m)
        if m == 1:
            train_result.append(num)
        num += 1
    train.append(ls)
with open('semeion_test.csv') as csvfile:
```

```
    reader = csv.reader(csvfile)
    rows= [row for row in reader]
for i in rows:
    ls = []
    temp = i[0].split()
    for m in temp[:-10]:
        m = float(m)
        ls.append(m)
    num = 0
    for m in temp[-10:]:
        m = int(m)
        if m == 1:
            test_right.append(num)
        num += 1
    test.append(ls)
```

Define KNN algorithm

Use train data, test data, train reult and k as the input of function. Calculate the geometry distance of the element to be identified, and choose the nearest k elements, which is used as the basis of classification.

```
def KNN(inX, train, train_result, k):
        size = len(train)
        train = np.asarray(train)
        inX = np.asarray(inX)
        result = []
        for X in inX:
            exp = np.tile(X, (size , 1))
            differ = exp - train
            square = differ ** 2
            distance = (square.sum(axis = 1)) ** 0.5
            # print(distance)
            sorted_index = distance.argsort()
            temp = [0] * 10
            for m in sorted_index[:k]:
                temp[train_result[m]] += 1
                temp = np.asarray(temp)
            result.append(temp.argsort()[-1])
        return result
```

Calculate the Accuracy

```
   result1 = KNN(test, train, train_result, 1)
   result3 = KNN(test, train, train_result, 3)
   result5 = KNN(test, train, train_result, 5)
   # calculate similarity
   def simrate(ls1, ls2):
       num = 0
       l = len(ls1)
       for i in range(l):
```

```
            if ls1[i] == ls2[i]:
                num += 1
        return format(num / l, '.2%')
    print("k = 1 similarity: ", simrate(result1, test_right))
    print("k = 3 similarity: ", simrate(result3, test_right))
    print("k = 5 similarity: ", simrate(result5, test_right))
    # compare with sklearn
    from sklearn.neighbors import KNeighborsClassifier
    knn1 = KNeighborsClassifier(1)
    knn1.fit(train, train_result)
    knn3 = KNeighborsClassifier(3)
    knn3.fit(train, train_result)
    knn5 = KNeighborsClassifier(5)
    knn5.fit(train, train_result)
    resultsk1 = knn1.predict(test)
    resultsk3 = knn3.predict(test)
    resultsk5 = knn5.predict(test)
    print("sklearn中k = 1 similarity: ", simrate(resultsk1, test_right))
    print("sklearn中k = 3 similarity: ", simrate(resultsk3, test_right))
    print("sklearn中k = 5 similarity: ", simrate(resultsk5, test_right))
```

```
knn中k = 1时的准确率是:  85.56%
knn中k = 3时的准确率是:  83.89%
knn中k = 5时的准确率是:  83.26%
sklearn中k = 1时的准确率是:  85.56%
sklearn中k = 3时的准确率是:  84.10%
sklearn中k = 5时的准确率是:  83.89%
```

It can be seen that:

When k = 1, the accuracy of the model from sklearn library and built from scratch are close.

When k = 3, the accuracy of the model from sklearn library is slightly higher than the model built from scratch.

When k = 5, the accuracy of the model from sklearn library is slightly higher than the model built from scratch.

Conclusion:

1. The accuracy of knn is slightly lower than the accuracy of the model from sklearn library.
2. The accuracy of both models drops slightly as k increases.

Handwritten Digit Recognition with LeNet

Experiment Description

使用`MNIST`手写数字体数据集进行训练和预测，实现测试集准确率达到98%及以上。本实验的目的：

- 掌握卷积神经网络基本原理，以`LeNet`为例
- 掌握主流框架的基本用法以及构建卷积神经网络的基本操作
- 了解如何使用`GPU`

Experiment Procedure

Preparation

Import libraries: pytorch, matplotlib, numpy, torchinfo. In MacOS, GPU acceleration is achieved through MPS of pytorch. Check if MPS is available.

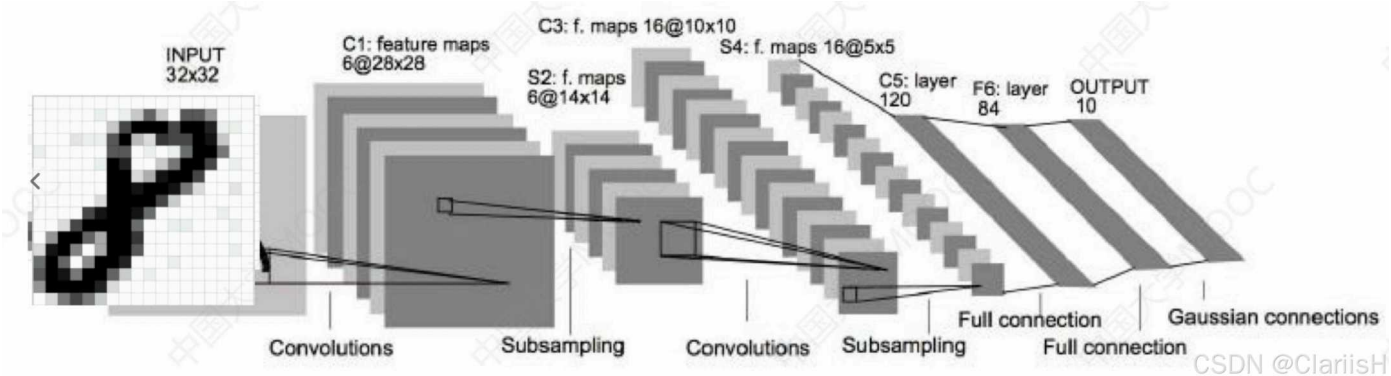
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import torchvision
import numpy as np
import torch.nn.functional as F
from torchinfo import summary
import os

# 检查 MPS
device = torch.device("mps")
print(torch.backends.mps.is_available())
```

Download Dataset

- Set the download path as data file under the project: use os.path.join to create new *data* folder as data_path to store train data and test data.
- Load MNIST dataset:
  - Use torchvision.transforms.ToTensor() as transformation function. Use datasets to download MNIST dataset to data_path, and transform it into a tensor(tensor is like ndarray, without dimention limit, suitable for deep learning matrix operation)
  - Set `batch_size` = 32
- Dataloader: use data loader `train_dl` and `test_dl` to load dataset into model in `batch_size`
- Check the data of first batch: Use `next(iter(train_dl))` to check the first batch of data in `train_dl`, and return `Batch shape: torch.Size([32, 1, 28, 28])`, which means load 32 data per batch, one channel, the height is 28px and the width is 32px.
- Show sample picture: show the first 20 sample pictures.

```
# 下载路径
data_path = './data'

# load data
transform = torchvision.transforms.ToTensor()
train_ds = torchvision.datasets.MNIST(data_path, train=True, transform=transform,
download=True)
test_ds = torchvision.datasets.MNIST(data_path, train=False, transform=transform,
download=True)

batch_size = 32

# dataloader
train_dl = torch.utils.data.DataLoader(train_ds, batch_size=batch_size, shuffle=True)
test_dl = torch.utils.data.DataLoader(test_ds, batch_size=batch_size)

# see first batch data
imgs, labels = next(iter(train_dl))
print("Batch shape:", imgs.shape)

# see pictures
plt.figure(figsize=(20, 5))
for i, img in enumerate(imgs[:20]):
    npimg = np.squeeze(img.numpy())
    plt.subplot(2, 10, i + 1)
    plt.imshow(npimg, cmap=plt.cm.binary)
    plt.axis('off')
plt.show()
```

Build the Model Network



The structure of the network.

Define `num_classes` as 10. The network model inherit nn.Module of pytorch.

Initialize Network Structure

- First convolusion layer: input 1 channel, and output 6 channels
- Second convolusion layer: input 6 channels, and output 16 channels
- Full connection layer

- fc1 input 16 5*5 features, and output 120 units.
- fc2 input 120 units, and output 84 units
- fc2 input 84 units, and output 10 categories(MNIST)

Forward：from input data x

- First layer of convolution + pooling + activation: After conv1 and relu, use F.max_pool2d(x, 2) for 2*2 max pooling
- Second layer of convolution + pooling + activation: After con2 and relu, use F.max_pool2d(x, 2) for 2*2 max pooling
- Flatten the data: torch.flatten(x, 1) flatten the data into one-dimension
- Full connection layer:
  - Pass the data into fc1, and the output function is relu.
  - Pass the data into fc2, and the output function is relu.
  - Pass the data into fc3, and get the final output.

```
# model definiton
num_classes = 10
import torch.nn.functional as F

class LeNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)

        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        # flatten
        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)  # output
        return x
```

Load and Print

Use .to(device) to transfer model to GPU. Use summary(model) to print the parameters of the model.

```
model = LeNet().to(device)
summary(model)
```

Model Training

1. Set hyperparameters

   Use cross entropy as the loss function of this classification model, set the learning rate to 1e-2 (0.01), and use stochastic gradient descent algorithm as optimization to update the parameters.

```
loss_fn = nn.CrossEntropyLoss()
learn_rate = 1e-2
opt = torch.optim.SGD(model.parameters(), lr=learn_rate)
```

2. Training function: train()

Firstly, calculate size and num_batches to use for later calculation of accuracy and loss of train dataset. Then, load the data in batches to GPU. Use trained model to forward the input data X, and get the prediction value pred. After that, use loss_fn to calculate the error between pred(predicted label) the y(real label), clear gradient, and calculate the new gradient of the loss to all parameters of the model. Finally, update the parameters of the model.

Change the boolean value represents whether the predicted label equals real label into float, and calculate training accuracy: Add the loss of all batches, calculate the sum loss, and output accuracy and loss.

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    train_loss, train_acc = 0, 0
    for X, y in dataloader:
        X, y = X.to(device), y.to(device) # move data to GPU
        pred = model(X)
        loss = loss_fn(pred, y) # calculate loss between predicted and real labels
        optimizer.zero_grad() # empty gradient
        loss.backward() # calculate gradient of all parameters
        optimizer.step() # update parameters
        train_acc += (pred.argmax(1) == y).type(torch.float).sum().item()
        train_loss += loss.item()
    return train_acc / size, train_loss / num_batches
```

Test Function

With torch.no_grad(): Forbid gradient calculation, which can save memory and compute resourse, because testing phase does not need backpropagation and parameters update. Calculate size and num_batches to facilitate the future calculation of accuracy and loss. Then load imgs and target data into GPU. Firstly use the trained model to forward the input `imgs`, and get the prediction `target_pred`. Use loss_fn to calculate the loss between target_pred and real label target. Use test_loss to accumulate loss. Finally, change the boolean value represents whether the predicted label equals real label into float, and calculate teat accuracy.

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, test_acc = 0, 0
    with torch.no_grad():
        for imgs, target in dataloader:
            imgs, target = imgs.to(device), target.to(device)
            target_pred = model(imgs)
            loss = loss_fn(target_pred, target)
            test_loss += loss.item()
            test_acc += (target_pred.argmax(1) == target).type(torch.float).sum().item()
    return test_acc / size, test_loss / num_batches
```

Training

Use two methods for training, with and without Batch Normalization and Dropout.

1. Set the epoch is 8, and train in batches, formatted output Epoch，Train_acc，Train_loss，Test_acc and Test_loss every batch.

```
# train
epochs = 8
train_loss = []
train_acc = []
test_loss = []
test_acc = []

for epoch in range(epochs):
    # model.train() # Use Batch Normalization, Dropout
    # epoch_train_acc, epoch_train_loss = train(train_dl, model, loss_fn, opt)
    # model.eval()
    epoch_test_acc, epoch_test_loss = test(test_dl, model, loss_fn)

    train_acc.append(epoch_train_acc)
    train_loss.append(epoch_train_loss)
    test_acc.append(epoch_test_acc)
    test_loss.append(epoch_test_loss)

    template = 'Epoch:{:2d}, Train_acc:{:.1f}%, Train_loss:{:.3f}, Test_acc:{:.1f}%,
Test_loss:{:.3f}'
    print(template.format(epoch + 1, epoch_train_acc * 100, epoch_train_loss,
epoch_test_acc * 100, epoch_test_loss))

print('Done')
```

Result Presentation

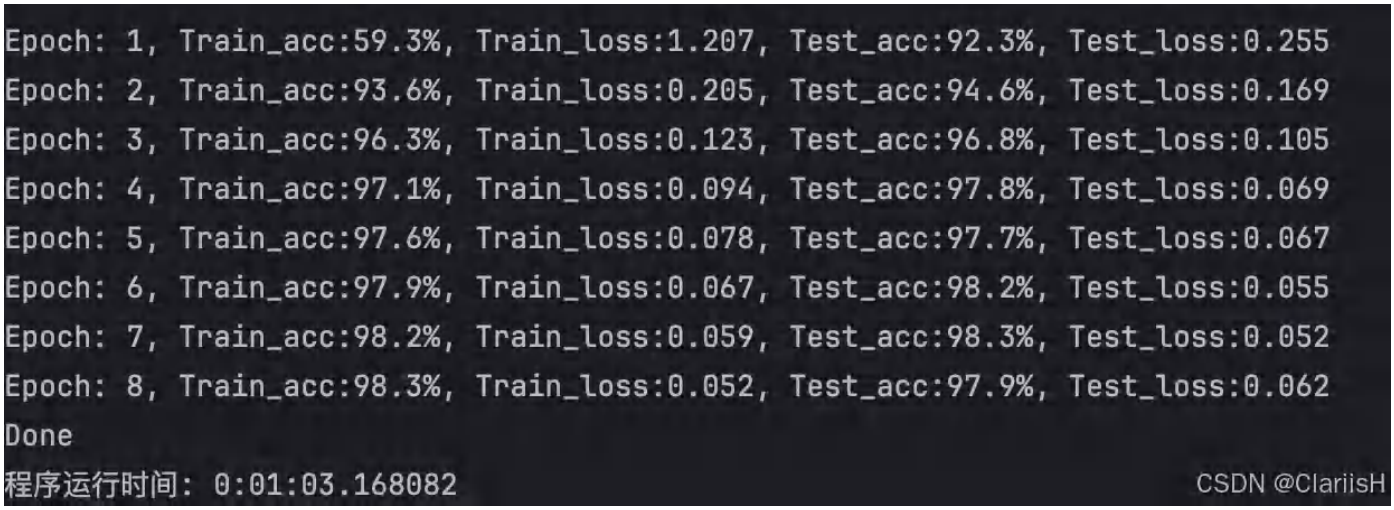
Plot Training/Test Accuracy plot and Training/Test Loss plot in the same line.

```
plt.rcParams['axes.unicode_minus'] = False
plt.rcParams['figure.dpi'] = 100

epochs_range = range(epochs)
plt.figure(figsize=(12, 3))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, train_acc, label='Training Accuracy')
plt.plot(epochs_range, test_acc, label='Test Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, train_loss, label='Training Loss')
plt.plot(epochs_range, test_loss, label='Test Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

Analysis

Print result

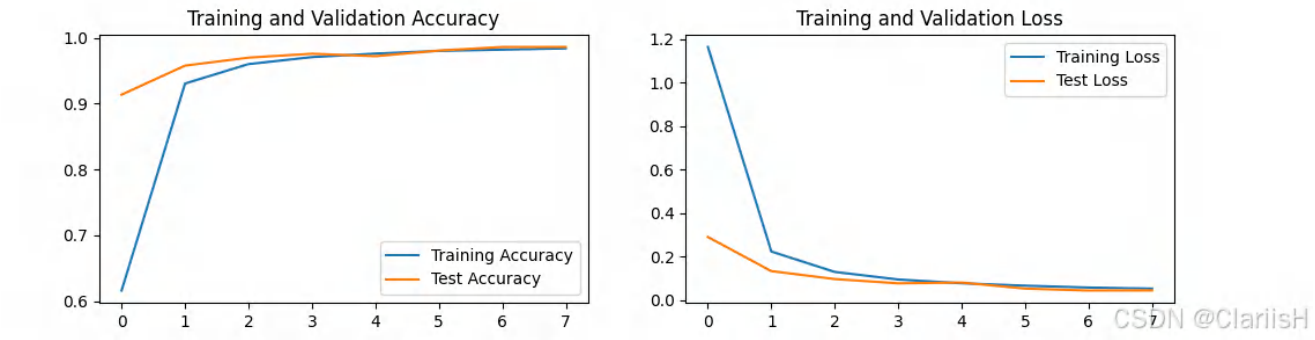
Without Batch Normalization and Dropout



Using Batch Normalization and Dropout

```
Epoch: 1, Train_acc:91.5%, Train_loss:0.279, Test_acc:94.7%, Test_loss:0.172
Epoch: 2, Train_acc:96.8%, Train_loss:0.104, Test_acc:97.3%, Test_loss:0.082
Epoch: 3, Train_acc:97.8%, Train_loss:0.070, Test_acc:98.1%, Test_loss:0.058
Epoch: 4, Train_acc:98.3%, Train_loss:0.054, Test_acc:98.4%, Test_loss:0.048
Epoch: 5, Train_acc:98.6%, Train_loss:0.045, Test_acc:98.2%, Test_loss:0.052
Epoch: 6, Train_acc:98.8%, Train_loss:0.037, Test_acc:98.5%, Test_loss:0.043
Epoch: 7, Train_acc:99.1%, Train_loss:0.031, Test_acc:98.7%, Test_loss:0.038
Epoch: 8, Train_acc:99.2%, Train_loss:0.027, Test_acc:98.7%, Test_loss:0.036
Done
程序运行时间: 0:02:00.865822
```


Using , the Training/Test Accuracy and Training/Test Loss plots.



It can be seen from the result that, the train result is good, the the accuracy is all above 95%. In this case, the accuracy of training increases rapidly, and the models have good performance on both train dataset and test dataset. If Batch Normalization and Dropout are implemented, the model reaches 98% of accuracy in the third epoch; if not implemented, it reaches 98% of accuracy in the sixth epoch. For the final result, the accuracy of using Batch Normalization and Dropout is also higher, but with a longer training time. Thus, it can be seen that Batch Normalization and Dropout can increase the accuracy of handwritting recognization by LeNet, with a higher final accuracy, but it also consumes more training time.