Final

20989977 Zhang Mingtao

2023/12/8

0.

```
library(reticulate)
```

1.

```
#1
# Random Forest:
#
# The idea of random forest: On the basis of bagging (changing training samples), the diversity of the base learner is further enhanced
# by changing the modeling variables. Specifically: for each node of the base decision tree, from the change of the node randomly
# select a subset containing k variables from the quantity set, and then select an optimal variable from this subset for branching.
#
# For each tree i = 1, • • • ,T: (1) Use the Bootstrap method to extract n sample observations from all training sample observations to
# form the Bootstrap data set D*; (2) Based on data set D* Construct a tree hi and repeat the following steps for each node in the tree
# until the stopping rule is met; (3) Output a combination of T trees.
#
# The computational complexity of random forest is: T(O(nk log2(n)) + O(s) + O(n))
# The computational complexity of the base decision tree is O(nk log2(n)); The complexity of Bootstrap sampling and voting/averaging
# is O(s); variables are randomly selected at the root node and intermediate nodes, with about n nodes, Therefore the complexity is O(n);
# There are T base decision trees in total.

# Gradient Boosting Trees:
#
# Decision Tree (GBDT) is an additive model form: fm(x) = fm-1(x) + hm(x).
# Consider the squared loss function, The hm(x) generated at step m should be in the direction of the local maximum decrease of L
# with respect to fm-1(x). In summary, at the mth step: hm(x) should be in the local direction described by the gradient
# -gm = y - fm-1(x) up. hm(x) should be a decision tree with εm = y - fm-1(x) as the dependent variable.
#
# The computational complexity of the decision tree can be expressed as O(TNMlog(M)), where T is the number of iterations.

#2
# Decision trees have strong interpretability, while random forests are relatively weak in model interpretability.

# Decision tree is a machine learning algorithm based on tree structure. Each node of the decision tree represents a feature
# attribute, the branches of the node represent the value of the feature attribute, and the leaf nodes represent the final
# classification or regression results. Due to the clear structure, we can directly observe the judgment conditions and branch
# paths of each node to understand how the model makes predictions.

# Random forest is an ensemble learning method that consists of multiple decision trees. The final prediction result of
# a random forest is obtained by voting or averaged by all decision trees. Each decision tree may adopt different features
# and parameter settings, so the interpretability of the entire model becomes more difficult. In addition, random forest
# introduces randomness in the construction process, including random selection of features and
```

```
random sampling of data,
# which also increases the difficulty of fully interpreting the model. Since the number of deci
sion trees in a random forest
# is large and the contribution of each decision tree is relatively small, it is difficult to m
ap the prediction results of
# the entire random forest to a single feature or decision.

#3
# Construct a Lagrangian function: Lagrange = L(Y, f) + λ(f1 + f2 + ... + fK), minimize Lagran
ge.
# We take the partial derivatives of f1, f2, ..., fK and λ and set them equal to zero to get t
he following system of equations:
#
# exp((-Y•f*)/K) • (-Y1/K) + λ = 0
# exp((-Y•f*)/K) • (-Y2/K) + λ = 0
# ...
# exp((-Y•f*)/K) • (-YK/K) + λ = 0
# f*1 + f*2 + ... + f*K = 0
#
# We can use numerical optimization methods to approximate the solution, such as gradient desce
nt.
# The class probability can be expressed as P(Y = 1 | G = Gk) = P(G = Gk). Yk = 1 or Yk = -1/(K
-1). therefore:
# P(Y = 1 | G = Gk) = P(G = Gk) = (1 + 1/(K-1)) * P(Yk = 1)
# P(Y = -1/(K-1) | G = Gk) = (1/(K-1)) * P(Yk = -1/(K-1))
# By definition we have ΣP(Yk = 1) = 1, therefore ΣP(G = Gk) = 1.
#
# When we minimize the loss function L(Y, f), the value of the loss function increases for misc
lassified samples and decreases
# for correctly classified samples. This causes the weight of incorrectly classified samples to
increase and the weight of
# correctly classified samples to decrease in the next iteration.
# Similar to Adaboost, we can calculate the weighting factor for each sample based on the class
ification error. Specifically,
# for sample i, we define the weight factor as: wi = exp((-Yi•f)/K). This weight factor is con
sistent with the form of
# the loss function L(Y, f)

#4
# Suppose there is an optimal classification hyperplane whose distance to the left is greater t
han the distance to the right.
# In this case, consider two projection points on the optimal classification hyperplane, which
are located on the boundaries of the left and right intervals respectively.
# Let the projection point on the left be A and the projection point on the right be B.
# Since the distance on the left side is the largest distance on the right side, we can move th
e projection point A on the optimal
# classification hyperplane along the direction of the normal arrangement, and at the same time
move the projection point B to the
# right until they are both located on their respective boundaries, instead of changing Classif
ication results of data points.
# Doing this will cause the method of optimizing the classification hyperplane to change, but s
ince we only made small adjustments,
# this new hyperplane will still be able to correctly classify the data points of both categori
es.
# However, this contradicts the definition of a maximum margin classifier.
```

```
#5
# Contains three types of support vectors:
# 1.Points lying on hyperplanes L+1 and L-1.  (0 < λi < C and ξi = 0 );
#
# 2.Points that fall within the interval and are correctly classified.   (λi = C and 0 < ξi ≤ 1);
#
# 3.Points that are not correctly classified.   (λi = C and ξi > 1).

#6
# From the perspective of loss function plus penalty: ξi can be expressed as: ξi = max(0,1-yi(β^T * xi + β0)). This is Hinge Loss.
# Using hinge loss, the above objective can be rewritten as: min(β,β0){1Σn(1-yi(xi^T *β + β0)) + λ/2|β|^2))}

#7
# The fundamental difference between the two algorithms is that K-means is essentially unsupervised learning,
# while KNN is supervised learning; K-means is a clustering algorithm, and KNN is a classification (or regression) algorithm.
#
# KNN belongs to supervised learning, and the categories are known. By training and learning the data of known categories,
# we can find the characteristics of these different categories, and then classify the unclassified data.
#
# Kmeans belongs to unsupervised learning. It is not known in advance how many categories the data will be divided into,
# and the data is aggregated into several groups through cluster analysis. Clustering does not require training and learning from the data.

#8
# Principal components analysis is an unsupervised technique that projects raw data into several high vertical directions
# These high vertical directions are orthogonal, so the correlation of the projected data is very low or almost close to 0.
# These feature transformations are linear.

# An autoencoder is an unsupervised artificial neural network that compresses data into lower dimensions and then reconstructs
# the input. Autoencoders find lower-dimensional representations of data by removing noise and redundancy on important features.

# PCA can only perform linear transformations, while autoencoders can perform both linear and nonlinear transformations;

# The PCA algorithm is fast to calculate, while the autoencoder needs to be trained through the gradient descent algorithm,
# so it takes longer time;

# PCA projects the data into several orthogonal directions, while the data dimensions are not necessarily orthogonal
# after autoencoder dimensionality reduction;

# The only hyperparameter of PCA is the number of orthogonal vectors, while the hyperparameters of the autoencoder are
```

```
# the structural parameters of the neural network;

# Autoencoders can also be used on complex, large data sets.

#9
# Bias: As the number of layers of a neural network increases, the complexity of the model incr
eases and it usually fits
# the training data better, so the bias gradually decreases. Deeper networks can learn more com
plex features and patterns,
# thereby increasing the flexibility and expressiveness of the model.

# Variance: When the number of layers of a neural network increases, the complexity of the mode
l also increases, which may
# lead to overfitting to the training data. Overfitting means that the model adapts too well to
the details and noise of the
# training data, resulting in reduced generalization ability on new unseen data. Therefore, the
variance may increase.

#10

# Input: training set D ={(x_n,y_m, verification set V, learning rate α, regularization coeffi
cient λ, number of network layers L,
#         number of neurons M_l,1<=l<=L

# Randomly initialize W,b;
# Repeat:
#        Randomly reorder the samples in the training set D;
#        for n = 1...N do:
#              Select samples (x_n,y_n) from the training set D;
#              Feedforward calculates the net input z_l and activation value a_l of each layer u
ntil the last layer;
#              Back propagation calculates the error δ_l of each layer;
#              Calculate the derivative of each layer parameter;
#              # Any l, dL(y_n,y^_n)/dW_l = δ_l•(a_(l-1))^T;
#              # Any l, dL(y_n,y^_n)/db_l = δ_l;
#              Update parameters;
#              # W_l ← W_l - α(δ_l•(a_(l-1))^T + λW_l);
#              # b_l ← b_l - α(δ_l);
#        end;
# Until the error rate of the neural network model on the validation set V no longer decreases.

# Output W,b
```

2.

```
#(2)
#1
documents=read.table("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project\\20newsgroup\\documents.txt", header = FALSE)
groupnames=read.table("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project\\20newsgroup\\groupnames.txt", header = FALSE)
newsgroups=read.table("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project\\20newsgroup\\newsgroups.txt", header = FALSE)
wordlist=read.table("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project\\20newsgroup\\wordlist.txt", header = FALSE)

library(tidyr)
library(gtools)

tent=pivot_wider(documents, names_from = V2, values_from = V3, values_fill = 0)
df=as.data.frame(tent)
df=df[, -1]
sorted_cols=mixedsort(colnames(df))
df=df[,sorted_cols]

colnames(df)=wordlist$V1

newsgroups[newsgroups == 1]=groupnames[1,]
newsgroups[newsgroups == 2]=groupnames[2,]
newsgroups[newsgroups == 3]=groupnames[3,]
newsgroups[newsgroups == 4]=groupnames[4,]

df=as.data.frame(lapply(df, as.factor))

df$grouptype=newsgroups$V1

df$grouptype=as.factor(df$grouptype)

library(randomForest)
```

```
## randomForest 4.7-1.1
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
library(caret)
```

```
## 载入需要的程辑包：ggplot2
```

```
##
## 载入程辑包：'ggplot2'
```

```
## The following object is masked from 'package:randomForest':
##
##     margin
```

```
## 载入需要的程辑包：lattice
```

```
train_control=trainControl(method = "cv",number = 5)
param_grid=expand.grid(mtry = c(8, 10, 12))

set.seed(123)
rf_model1=train(x = df[,-101],y = df[,101], method = "rf", ntree = 150, trControl = train_contr
ol, tuneGrid = param_grid)
rf_model1$results
```

```
##   mtry  Accuracy     Kappa  AccuracySD     KappaSD
## 1    8 0.8121536 0.7405424 0.006893363 0.009659618
## 2   10 0.8128924 0.7417440 0.005680786 0.007970848
## 3   12 0.8132002 0.7423514 0.006341950 0.008816919
```

```
set.seed(123)
rf_model2=train(x = df[,-101],y = df[,101], method = "rf", ntree = 100, trControl = train_contr
ol, tuneGrid = param_grid)
rf_model2$results
```

```
##   mtry  Accuracy     Kappa  AccuracySD     KappaSD
## 1    8 0.8117226 0.7399285 0.006920613 0.009678856
## 2   10 0.8120920 0.7406228 0.005971920 0.008345824
## 3   12 0.8128924 0.7419563 0.006409030 0.008845819
```

```
set.seed(123)
rf_model3=train(x = df[,-101],y = df[,101], method = "rf", ntree = 200, trControl = train_contr
ol, tuneGrid = param_grid)
rf_model3$results
```

```
##   mtry  Accuracy     Kappa  AccuracySD     KappaSD
## 1    8 0.8118457 0.7400971 0.006960800 0.009816271
## 2   10 0.8135697 0.7427086 0.005365292 0.007505029
## 3   12 0.8148625 0.7446613 0.005050184 0.007068843
```

```
# We choose the ntree = 200 and mtry = 12 to get the lowest cv-error = 1 - 0.8148625 = 0.185137
5

set.seed(123)
rf_model=randomForest(grouptype~., data = df, mtry=12, ntree=200, importance=T, proximity=T)
rf_model
```

```
##
## Call:
##  randomForest(formula = grouptype ~ ., data = df, mtry = 12, ntree = 200,      importance = T, proximity = T)
##                Type of random forest: classification
##                      Number of trees: 200
## No. of variables tried at each split: 12
##
##         OOB estimate of  error rate: 18.65%
## Confusion matrix:
##        comp.* rec.* sci.* talk.* class.error
## comp.*   4142    73   195    195   0.1005429
## rec.*     300  2706   156    357   0.2310315
## sci.*     642   131  1488    396   0.4399699
## talk.*    258   126   200   4877   0.1069401
```

```
# OOB estimate of  error rate: 18.65%
# Confusion matrix:
#          comp.* rec.* sci.* talk.* class.error
# comp.*   4142    73   195    195   0.1005429
# rec.*     300  2706   156    357   0.2310315
# sci.*     642   131  1488    396   0.4399699
# talk.*    258   126   200   4877   0.1069401

sorted_MeanDecreaseAccuracy=rf_model$importance[order(rf_model$importance[,5], decreasing = TRUE), ]
sorted_MeanDecreaseGini=rf_model$importance[order(rf_model$importance[,6], decreasing = TRUE), ]

sorted_MeanDecreaseAccuracy[1:10,]  # the same ↓
```

```
##                comp.*      rec.*      sci.*      talk.* MeanDecreaseAccuracy
## windows    0.03229535 0.03096465 0.03063397 0.031127670           0.03133656
## car        0.02791481 0.07113461 0.01662072 0.014315823           0.03084363
## god        0.03517044 0.02235538 0.02462732 0.021447018           0.02605346
## christian  0.02537361 0.02098339 0.02713232 0.024058559           0.02426748
## government 0.03195486 0.02320699 0.01459359 0.017853451           0.02247704
## team       0.02139258 0.01646405 0.01114233 0.018034239           0.01751500
## space      0.01090275 0.01251518 0.04764293 0.008482974           0.01645380
## jews       0.02299562 0.01843499 0.01760104 0.007508724           0.01591215
## graphics   0.01750839 0.01383242 0.01114834 0.014459737           0.01464551
## religion   0.02033723 0.01474761 0.01641379 0.006120233           0.01369789
##            MeanDecreaseGini
## windows            529.4252
## car                350.6786
## god                400.1818
## christian          383.6470
## government         334.6910
## team               285.7007
## space              196.7427
## jews               252.1002
## graphics           222.5499
## religion           190.3336
```

```
sorted_MeanDecreaseGini[1:10,]      # the same ↑
```

```
##                 comp.*      rec.*      sci.*      talk.* MeanDecreaseAccuracy
## windows    0.03229535 0.03096465 0.03063397 0.031127670           0.03133656
## god        0.03517044 0.02235538 0.02462732 0.021447018           0.02605346
## christian  0.02537361 0.02098339 0.02713232 0.024058559           0.02426748
## car        0.02791481 0.07113461 0.01662072 0.014315823           0.03084363
## government 0.03195486 0.02320699 0.01459359 0.017853451           0.02247704
## team       0.02139258 0.01646405 0.01114233 0.018034239           0.01751500
## jews       0.02299562 0.01843499 0.01760104 0.007508724           0.01591215
## graphics   0.01750839 0.01383242 0.01114834 0.014459737           0.01464551
## space      0.01090275 0.01251518 0.04764293 0.008482974           0.01645380
## religion   0.02033723 0.01474761 0.01641379 0.006120233           0.01369789
##            MeanDecreaseGini
## windows            529.4252
## god                400.1818
## christian          383.6470
## car                350.6786
## government         334.6910
## team               285.7007
## jews               252.1002
## graphics           222.5499
## space              196.7427
## religion           190.3336
```

```
# So the ten most important keywords based on variable importance are:
# windows, god, christian, car, government, team, jews, graphics, space, religion.




#2
train_control2=trainControl(method = "cv", number = 5)
param_grid2=expand.grid(n.trees = c(100, 150, 200),interaction.depth = c(1,2,3),shrinkage = c
(0.01,0.05,0.1),n.minobsinnode = c(15))

set.seed(123)
gbm_model=train(x = df[, -101],y = df[, 101],method = "gbm",trControl = train_control2,tuneGrid
= param_grid2,verbose = FALSE)

gbm_model$results
```

```
##    shrinkage interaction.depth n.minobsinnode n.trees  Accuracy     Kappa
## 1       0.01                 1             15     100 0.5738208 0.3844925
## 10      0.05                 1             15     100 0.7482448 0.6468091
## 19      0.10                 1             15     100 0.7861719 0.7026985
## 4       0.01                 2             15     100 0.6771327 0.5421739
## 13      0.05                 2             15     100 0.7864178 0.7026910
## 22      0.10                 2             15     100 0.8058736 0.7323159
## 7       0.01                 3             15     100 0.7055779 0.5845913
## 16      0.05                 3             15     100 0.7996549 0.7228541
## 25      0.10                 3             15     100 0.8106758 0.7392044
## 2       0.01                 1             15     150 0.6165496 0.4492386
## 11      0.05                 1             15     150 0.7728727 0.6827977
## 20      0.10                 1             15     150 0.7997165 0.7229461
## 5       0.01                 2             15     150 0.7036076 0.5813294
## 14      0.05                 2             15     150 0.8000244 0.7233939
## 23      0.10                 2             15     150 0.8104296 0.7389196
## 8       0.01                 3             15     150 0.7447971 0.6420190
## 17      0.05                 3             15     150 0.8083978 0.7358106
## 26      0.10                 3             15     150 0.8122766 0.7416213
## 3       0.01                 1             15     200 0.6587852 0.5132910
## 12      0.05                 1             15     200 0.7872184 0.7039172
## 21      0.10                 1             15     200 0.8045805 0.7304356
## 6       0.01                 2             15     200 0.7266958 0.6153271
## 15      0.05                 2             15     200 0.8059350 0.7323624
## 24      0.10                 2             15     200 0.8113531 0.7403126
## 9       0.01                 3             15     200 0.7568647 0.6593792
## 18      0.05                 3             15     200 0.8107376 0.7394059
## 27      0.10                 3             15     200 0.8132616 0.7430476
##     AccuracySD     KappaSD
## 1  0.009379922 0.014123813
## 10 0.009299975 0.013365184
## 19 0.006642032 0.009337847
## 4  0.011708544 0.017826167
## 13 0.007779747 0.010869979
## 22 0.007051318 0.009645575
## 7  0.009387068 0.013723561
## 16 0.007656894 0.010703235
## 25 0.005306819 0.007233715
## 2  0.012705015 0.019029272
## 11 0.008569324 0.012332924
## 20 0.008065616 0.011319931
## 5  0.007008193 0.010043459
## 14 0.006944677 0.009619840
## 23 0.004912889 0.006708386
## 8  0.006383247 0.009314182
## 17 0.004517136 0.006220609
## 26 0.004486879 0.006162411
## 3  0.014277344 0.021274373
## 12 0.007984490 0.011382984
## 21 0.006680739 0.009221109
## 6  0.010053031 0.014611909
## 15 0.004517278 0.006021629
## 24 0.004953803 0.006814072
## 9  0.008019587 0.011619573
```

```
## 18 0.005143209 0.007052146
## 27 0.006182338 0.008587141
```

```
gbm_model
```

```
## Stochastic Gradient Boosting
##
## 16242 samples
##   100 predictor
##     4 classes: 'comp.*', 'rec.*', 'sci.*', 'talk.*'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 12993, 12994, 12994, 12993, 12994
## Resampling results across tuning parameters:
##
##   shrinkage  interaction.depth  n.trees  Accuracy   Kappa
##   0.01       1                  100      0.5738208  0.3844925
##   0.01       1                  150      0.6165496  0.4492386
##   0.01       1                  200      0.6587852  0.5132910
##   0.01       2                  100      0.6771327  0.5421739
##   0.01       2                  150      0.7036076  0.5813294
##   0.01       2                  200      0.7266958  0.6153271
##   0.01       3                  100      0.7055779  0.5845913
##   0.01       3                  150      0.7447971  0.6420190
##   0.01       3                  200      0.7568647  0.6593792
##   0.05       1                  100      0.7482448  0.6468091
##   0.05       1                  150      0.7728727  0.6827977
##   0.05       1                  200      0.7872184  0.7039172
##   0.05       2                  100      0.7864178  0.7026910
##   0.05       2                  150      0.8000244  0.7233939
##   0.05       2                  200      0.8059350  0.7323624
##   0.05       3                  100      0.7996549  0.7228541
##   0.05       3                  150      0.8083978  0.7358106
##   0.05       3                  200      0.8107376  0.7394059
##   0.10       1                  100      0.7861719  0.7026985
##   0.10       1                  150      0.7997165  0.7229461
##   0.10       1                  200      0.8045805  0.7304356
##   0.10       2                  100      0.8058736  0.7323159
##   0.10       2                  150      0.8104296  0.7389196
##   0.10       2                  200      0.8113531  0.7403126
##   0.10       3                  100      0.8106758  0.7392044
##   0.10       3                  150      0.8122766  0.7416213
##   0.10       3                  200      0.8132616  0.7430476
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 15
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 200, interaction.depth =
##  3, shrinkage = 0.1 and n.minobsinnode = 15.
```

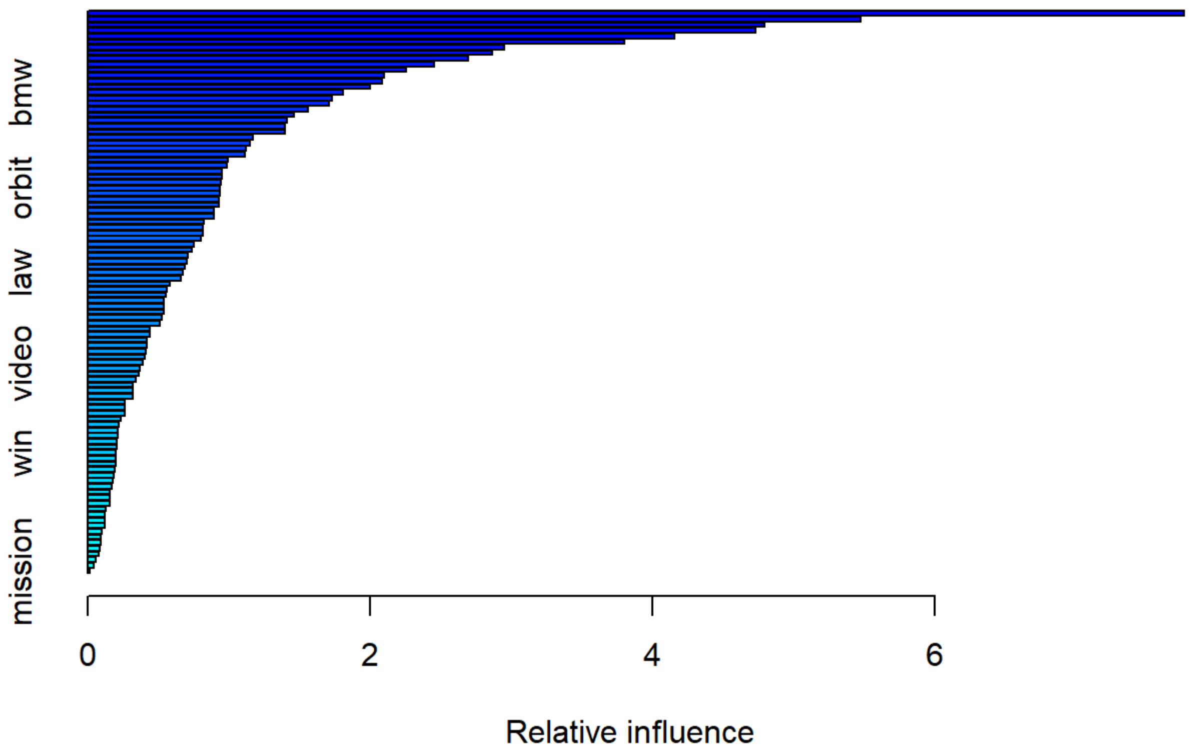
```
# The final values used for the model were n.trees = 200, interaction.depth = 3, shrinkage = 0.
1 and n.minobsinnode = 15.

library(gbm)
```

```
## Loaded gbm 2.1.8.1
```

```
set.seed(123)
gbm_model2=gbm(grouptype~., data = df, distribution = "multinomial",n.trees=200, interaction.de
pth=3, shrinkage = 0.1)
```

```
## Warning: Setting `distribution = "multinomial"` is ill-advised as it is
## currently broken. It exists only for backwards compatibility. Use at your own
## risk.
```

```
summary(gbm_model2)
```



```
##                   var    rel.inf
## windows       windows 7.76830423
## god               god 5.47813458
## car               car 4.79586424
## christian   christian 4.73438358
## government government 4.15566518
## team             team 3.80285496
## jews             jews 2.95343128
## graphics     graphics 2.86835188
## space           space 2.69446992
## gun               gun 2.45298056
## baseball     baseball 2.25918380
## religion     religion 2.10315069
## mac               mac 2.08606114
## hockey         hockey 2.00185731
## bmw               bmw 1.81238261
## games           games 1.72910012
## season         season 1.70978899
## card             card 1.56451244
## children     children 1.46171761
## players       players 1.41188087
## israel         israel 1.39942468
## software     software 1.39514316
## engine         engine 1.16907795
## honda           honda 1.15280929
## pc                 pc 1.12227389
## bible           bible 1.11413200
## jesus           jesus 0.99583387
## computer     computer 0.98733590
## evidence     evidence 0.95458364
## nasa             nasa 0.95313038
## doctor         doctor 0.94674961
## orbit           orbit 0.94013142
## president   president 0.93464514
## files           files 0.93027890
## medicine     medicine 0.92825594
## shuttle       shuttle 0.89728783
## dos               dos 0.89668126
## email           email 0.82305278
## disease       disease 0.81529309
## scsi             scsi 0.81320844
## war               war 0.80301519
## program       program 0.75604806
## health         health 0.73611568
## rights         rights 0.71038343
## disk             disk 0.70646825
## server         server 0.69038198
## law               law 0.67562059
## moon             moon 0.66066827
## help             help 0.57909248
## nhl               nhl 0.56424510
## memory         memory 0.55252587
## msg               msg 0.53861626
## format         format 0.53753358
## drive           drive 0.53727699
```

```
## fact             fact 0.52555104
## hit               hit 0.51373660
## league         league 0.44025515
## insurance   insurance 0.43769136
## patients     patients 0.42230895
## display       display 0.41981329
## image           image 0.41109948
## version       version 0.40751874
## video           video 0.38880402
## solar           solar 0.37027877
## problem       problem 0.36432840
## launch         launch 0.34028202
## ftp               ftp 0.32090847
## phone           phone 0.32058865
## fans             fans 0.31857985
## water           water 0.26355555
## power           power 0.26271650
## case             case 0.26235176
## system         system 0.23556983
## data             data 0.21973287
## cancer         cancer 0.21571308
## world           world 0.21140762
## research     research 0.20670492
## science       science 0.20561366
## win               win 0.20169982
## human           human 0.19686561
## dealer         dealer 0.19654442
## state           state 0.19317984
## food             food 0.18408411
## course         course 0.17797347
## satellite   satellite 0.17220098
## oil               oil 0.16026728
## won               won 0.15809727
## driver         driver 0.15545104
## puck             puck 0.13054977
## studies       studies 0.12428429
## lunar           lunar 0.12136138
## mars             mars 0.11892162
## question     question 0.09842825
## university university 0.09184271
## technology technology 0.09026141
## earth           earth 0.08279134
## aids             aids 0.08051708
## number         number 0.05425566
## vitamin       vitamin 0.04564852
## mission       mission 0.01826669
```

```
# So the ten most important keywords based on variable importance are:
# windows, god, christian, car, government, team, jews, graphics, space, gun (not religion).

predicted_classes=predict(gbm_model2, newdata = df, type = "response")
```

```
## Using 200 trees...
```

```
predicted_classes=colnames(predicted_classes)[apply(predicted_classes, 1, which.max)]

confusion_matrix=table(predicted_classes, df$grouptype)
confusion_matrix
```

```
##
## predicted_classes comp.* rec.* sci.* talk.*
##            comp.*   4142   275   553    231
##            rec.*      62  2739   117    111
##            sci.*     217   160  1622    257
##            talk.*    184   345   365   4862
```

```
# 计算错误率
error_rate=1 - sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Error rate:", error_rate))
```

```
## [1] "Error rate: 0.177133357960842"
```

```
# predicted_classes  comp.* rec.* sci.* talk.*
#           comp.*   4142   275   553    231
#           rec.*      62  2739   117    111
#           sci.*     217   160  1622    257
#           talk.*    184   345   365   4862
#
#            Error rate: 0.177133357960842



#3
#  Time : the gbm(Error rate: 0.177) is much slower and a bit more accurate than random forest
(Error rate: 0.1851375).
#  Variable importance: the first 9 keywords are the same.

#4
library(MASS)

ctrl=trainControl(method = "cv", number = 5,verboseIter = FALSE)
lda_model=train(grouptype ~ ., data = df, method = "lda", trControl = ctrl)
lda_model
```

```
## Linear Discriminant Analysis
##
## 16242 samples
##   100 predictor
##     4 classes: 'comp.*', 'rec.*', 'sci.*', 'talk.*'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 12992, 12995, 12994, 12993, 12994
## Resampling results:
##
##   Accuracy   Kappa
##   0.7976864  0.7213268
```

```
lda_model$results
```

```
##   parameter  Accuracy     Kappa  AccuracySD    KappaSD
## 1      none 0.7976864 0.7213268 0.008227586 0.01140499
```

```
# Accuracy: 0.7974388
# Misclassification Error = 1 - 0.7974388 = 0.2025612

#5
# I must reduce the dimensionality first otherwise qda will report an error: Error in qda.defau
lt(x, grouping, ...) : rank deficiency in group comp.*

df1=as.data.frame(tent)
df1=df1[, -1]
sorted_cols1=mixedsort(colnames(df1))
df1=df1[,sorted_cols1]
colnames(df1)=wordlist$V1

# df1=as.data.frame(lapply(df1, as.factor))

df1$grouptype=newsgroups$V1
df1$grouptype=as.factor(df1$grouptype)

# 进行主成分分析（PCA）
pca_result=prcomp(df1[, -which(names(df1) == "grouptype")], scale. = TRUE)  # 选择去除响应变量
后的预测变量列

# 选择保留的主成分数量或方差百分比
# 这里以保留方差百分比为例，比如保留累积方差达到90%的主成分
variance_threshold=0.9
cumulative_variance=cumsum(pca_result$sdev^2) / sum(pca_result$sdev^2)
num_components=which(cumulative_variance >= variance_threshold)[1]

# 使用选定的主成分数量进行降维
reduced_data=as.data.frame(predict(pca_result, newdata = df1)[, 1:num_components])

# 将响应变量添加回降维后的数据框
reduced_data$grouptype=df$grouptype

# 训练 QDA 模型
qda_model=train(grouptype ~ ., data = reduced_data, method = "qda", trControl = ctrl)
qda_model
```

```
## Quadratic Discriminant Analysis
##
## 16242 samples
##    84 predictor
##     4 classes: 'comp.*', 'rec.*', 'sci.*', 'talk.*'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 12994, 12993, 12992, 12994, 12995
## Resampling results:
##
##   Accuracy   Kappa
##   0.7712093  0.6839627
```

```
qda_model$results
```

```
##   parameter  Accuracy     Kappa  AccuracySD    KappaSD
## 1      none 0.7712093 0.6839627 0.008590708 0.01184045
```

```
# Accuracy: 0.7710264
# Misclassification Error = 1 - 0.7710264 = 0.2289736



#6
library(e1071)
```

```
##
## 载入程辑包：'e1071'
```

```
## The following object is masked from 'package:gtools':
##
##     permutations
```

```
tune_ctrl=tune.control(sampling = "cross", cross = 5)

set.seed(1)
tune.out=tune(svm, grouptype~.,data=df,kernel="linear",scale=TRUE,ranges=list(cost=c(1,5,10,15,20,25,30)),tunecontrol=tune_ctrl)
tune.out$performances
```

```
##   cost     error  dispersion
## 1    1 0.1918484 0.008922389
## 2    5 0.1912325 0.009696162
## 3   10 0.1911094 0.009961910
## 4   15 0.1919099 0.010134537
## 5   20 0.1916636 0.009597767
## 6   25 0.1920330 0.009913662
## 7   30 0.1913557 0.009783619
```

```
tune.out$best.performance
```

```
## [1] 0.1911094
```

```
tune.out$best.model
```

```
##
## Call:
## best.tune(method = svm, train.x = grouptype ~ ., data = df, ranges = list(cost = c(1,
##     5, 10, 15, 20, 25, 30)), tunecontrol = tune_ctrl, kernel = "linear",
##     scale = TRUE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  5712
```

```
# Accuracy: 0.8088906
# Misclassification Error = 0.1911094



# set.seed(1)
# tune.out=tune(svm, grouptype~.,data=df, kernel="radial",ranges=list(cost=c(0.1,1,10,100),gamma=c(0.5,1,2,3,4)),tunecontrol=tune_ctrl)
# summary(tune.out)



#7
#
#      MODEL        Accuracy   Time cost to train models
#
#  Random Forest   0.8148625     Middle
#       GBM        0.8228666     Large
#       LDA        0.7974388     Small
#       QDA        0.7710264     Small
#       SVM        0.8088906     Large

# The GBM has the best accuracy, however it needs the most time to train the model. Random Forest is better.
# The SVM also takes lots of time while its performance is not so good as Random Forest.

write.csv(df, file = "C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project\\group_data.csv", row.names = FALSE)
```

3.

```python
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
import math

#(3)
#1
def getEuclidean(point1, point2):
    dimension = len(point1)
    dist = 0.0
    for i in range(dimension):
        dist += (point1[i] - point2[i]) ** 2
    return math.sqrt(dist)

def k_means(df, k, iteration):
    #初始化簇心向量
    index = random.sample(list(range(len(df))), k)
    vectors = []
    for i in index:
        vectors.append(list(df.loc[i,].values))

    #初始化类别
    labels = []
    for i in range(len(df)):
        labels.append(-1)

    while(iteration > 0):
        #初始化簇
        C = []
        for i in range(k):
            C.append([])
        for labelIndex, item in enumerate(df.to_numpy()):
            classIndex = -1
            minDist = 1e6
            for i, point in enumerate(vectors):
                dist = getEuclidean(item, point)
                if(dist < minDist):
                    classIndex = i
                    minDist = dist
            C[classIndex].append(item)
            labels[labelIndex] = classIndex

        for i, cluster in enumerate(C):
            clusterHeart = []
            dimension = df.shape[1]
            for j in range(dimension):
                clusterHeart.append(0)
            for item in cluster:
                for j, coordinate in enumerate(item):
                    clusterHeart[j] += coordinate / len(cluster)
            vectors[i] = clusterHeart

        iteration -= 1
    return C, labels
```

```
#2
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
import time

df0 = pd.read_csv("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project\\group_data.csv")
df = df0.iloc[:,0:100]

scaler = StandardScaler()

# 对数据进行标准化
scaled_df = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)

start_time = time.time()

# 创建 PCA 模型并进行主成分分析
pca = PCA(n_components=4)
principal_components = pca.fit_transform(scaled_df)

# 将主成分数据转换为数据框
pc_df = pd.DataFrame(data=principal_components, columns=['PC1','PC2','PC3','PC4'])

random.seed(123)
C, labels = k_means(pc_df, 4, 20)

pc_df.loc[:,'grouptype']=labels

df0['grouptype'] = df0['grouptype'].replace('comp.*', 2)
df0['grouptype'] = df0['grouptype'].replace('talk.*', 3)
df0['grouptype'] = df0['grouptype'].replace('sci.*', 0)
df0['grouptype'] = df0['grouptype'].replace('rec.*', 1)

# 计算混淆矩阵
cm = confusion_matrix(df0.loc[:,'grouptype'], labels)
print(cm)

# 计算误判率
```

```
## [[ 343 1882  153  279]
##  [   0 3450   34   35]
##  [  10 2469 2121    5]
##  [  11 3661   14 1775]]
```

```
misclassification_rate = (np.sum(cm) - np.trace(cm)) / np.sum(cm)

print("misclassification_rate: "+ str(misclassification_rate))
```

```
## misclassification_rate: 0.5265977096416697
```

```
end_time = time.time()
execution_time = end_time - start_time
print(f"Total cost time: {execution_time:.4f} seconds")


#3
```

```
## Total cost time: 3.4608 seconds
```

```
df0 = pd.read_csv("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final proj
ect\\group_data.csv")
df = df0.iloc[:,0:100]

scaler = StandardScaler()

# 对数据进行标准化
scaled_df = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)

start_time = time.time()

# 创建 PCA 模型并进行主成分分析
pca = PCA(n_components=5)
principal_components = pca.fit_transform(scaled_df)

# 将主成分数据转换为数据框
pc_df = pd.DataFrame(data=principal_components, columns=['PC1','PC2','PC3','PC4','PC5'])

random.seed(123)
C, labels = k_means(pc_df, 4, 20)

pc_df.loc[:,'grouptype']=labels

df0['grouptype'] = df0['grouptype'].replace('comp.*', 2)
df0['grouptype'] = df0['grouptype'].replace('talk.*', 3)
df0['grouptype'] = df0['grouptype'].replace('sci.*', 0)
df0['grouptype'] = df0['grouptype'].replace('rec.*', 1)

# 计算混淆矩阵
cm = confusion_matrix(df0.loc[:,'grouptype'], labels)
print(cm)

# 计算误判率
```

```
## [[ 343 2024  144  146]
##  [   0 3459   25   35]
##  [  10 2506 2084    5]
##  [  11 3391   14 2045]]
```

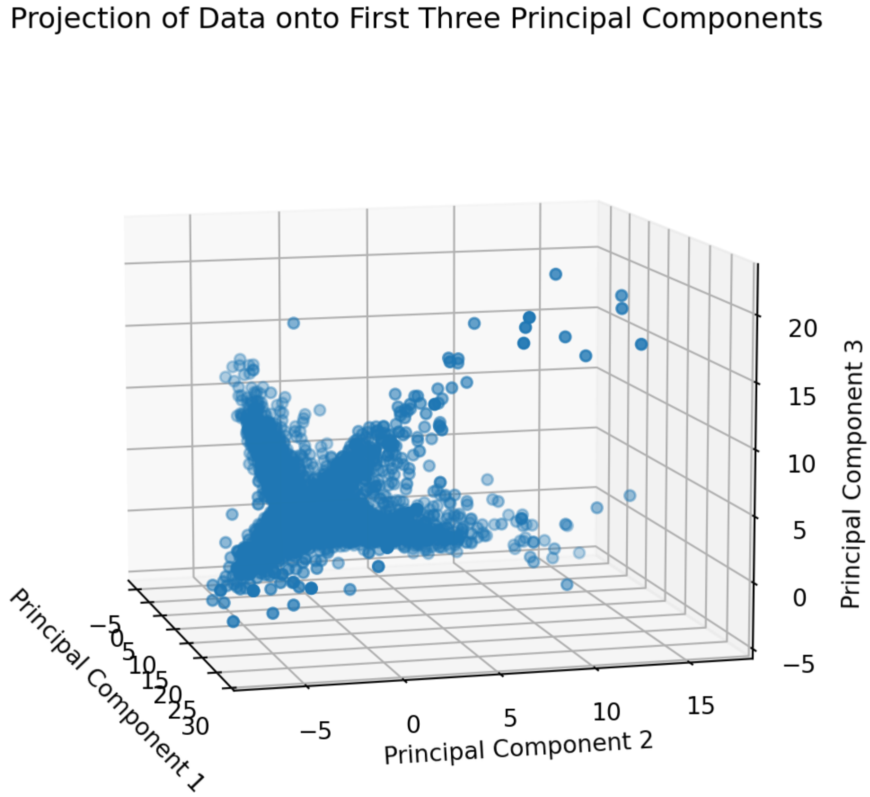
```
misclassification_rate = (np.sum(cm) - np.trace(cm)) / np.sum(cm)

print("misclassification_rate: "+ str(misclassification_rate))
```

```
## misclassification_rate: 0.5116980667405492
```

```
end_time = time.time()
execution_time = end_time - start_time
print(f"Total cost time: {execution_time:.4f} seconds")



#4
#      MODEL        Accuracy   Time cost to train models
#
#  Random Forest   0.8148625     Middle
#       GBM        0.8228666     Large
#       LDA        0.7974388     Small
#       QDA        0.7710264     Small
#       SVM        0.8088906     Large
#     K-means      0.4881788     Small



#5
# 使用 PCA 将数据投影到前三个主成分上
```

```
## Total cost time: 3.4262 seconds
```

```
pca = PCA(n_components=3)
df_pca = pca.fit_transform(scaled_df)

# 绘制投影图
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

# 绘制散点图
ax.scatter(df_pca[:, 0], df_pca[:, 1], df_pca[:, 2], marker='o')

ax.set_xlabel('Principal Component 1')
ax.set_ylabel('Principal Component 2')
ax.set_zlabel('Principal Component 3')
ax.set_title('Projection of Data onto First Three Principal Components')

ax.view_init(elev=10, azim=-15)

plt.show()

# We can see the 4 clusters' structure.
```


Projection of Data onto First Three Principal Components
Principal Component 1
−5
0
5
10
15
20
25
30
Principal Component 2
−5
0
5
10
15
Principal Component 3
−5
0
5
10
15
20

4.

```
#(4)
#1
test=read.csv("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project
\\MNIST\\test_resized.csv")
train=read.csv("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project
\\MNIST\\train_resized.csv")

# train[, 2:ncol(train)][train[, 2:ncol(train)] != 0]= 1
# test[, 2:ncol(test)][test[, 2:ncol(test)] != 0]= 1

train_36=train[train$label==3 | train$label==6,]
test_36=test[test$label==3 | test$label==6,]

train_36$label=as.factor(train_36$label)
test_36$label=as.factor(test_36$label)

library(e1071)
library(caret)

start_time=Sys.time()  # start time

tune_ctrl=tune.control(sampling = "cross", cross = 5)

set.seed(123)
tune.out=tune(svm, label~.,data=train_36,kernel="linear",scale=TRUE,ranges=list(cost=c(0.01,0.0
2,0.05,0.1,0.5,1,3,10)),tunecontrol=tune_ctrl)

end_time=Sys.time()    # end time
execution_time = end_time-start_time
execution_time
```

```
## Time difference of 34.0482 secs
```

```
tune.out$performances
```

```
##    cost       error  dispersion
## 1  0.01 0.005974003 0.001595727
## 2  0.02 0.005475802 0.002161809
## 3  0.05 0.006139978 0.002163735
## 4  0.10 0.006471790 0.002226099
## 5  0.50 0.008795167 0.003645619
## 6  1.00 0.009459067 0.003451907
## 7  3.00 0.009625042 0.003246368
## 8 10.00 0.009625042 0.003246368
```

```
tune.out$best.performance
```

```
## [1] 0.005475802
```

```
tune.out$best.model
```

```
##
## Call:
## best.tune(method = svm, train.x = label ~ ., data = train_36, ranges = list(cost = c(0.01,
##     0.02, 0.05, 0.1, 0.5, 1, 3, 10)), tunecontrol = tune_ctrl, kernel = "linear",
##     scale = TRUE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.02
##
## Number of Support Vectors:  182
```

```
# choose cost = 0.02

svmfit1=svm(label~., data=train_36, kernel="linear", cost=0.02 , scale=TRUE)
### prediction
ypred1=predict(svmfit1,test_36)
table(predict=ypred1, truth=test_36$label)  # confusion matrix
```

```
##        truth
## predict    3    6
##       3 1252    5
##       6   10 1195
```

```
sum(ypred1==test_36$label)/nrow(test_36)   # accuracy
```

```
## [1] 0.9939074
```

```
1-sum(ypred1==test_36$label)/nrow(test_36)  # the mis-classification error
```

```
## [1] 0.006092608
```

```
#2
start_time=Sys.time()  # start time

tune_ctrl=tune.control(sampling = "cross", cross = 5)

set.seed(123)
tune.out=tune(svm, label~.,data=train_36,kernel="radial",scale=TRUE,ranges=list(cost=c(0.5,1,4,
9),gamma=c(0.001,0.01,0.1,0.5)),tunecontrol=tune_ctrl)

end_time=Sys.time()    # end time
execution_time = end_time-start_time
execution_time
```

```
## Time difference of 21.48257 mins
```

```
tune.out$performances
```

```
##    cost gamma       error   dispersion
## 1   0.5 0.001 0.006471515 0.0032867076
## 2   1.0 0.001 0.005475802 0.0025960540
## 3   4.0 0.001 0.004646340 0.0009447263
## 4   9.0 0.001 0.004480502 0.0019102488
## 5   0.5 0.010 0.012944407 0.0027918781
## 6   1.0 0.010 0.009459342 0.0027283811
## 7   4.0 0.010 0.008795442 0.0023917095
## 8   9.0 0.010 0.008795580 0.0024630158
## 9   0.5 0.100 0.185862940 0.0083632723
## 10  1.0 0.100 0.132760953 0.0107151313
## 11  4.0 0.100 0.122970349 0.0117187260
## 12  9.0 0.100 0.122970349 0.0117187260
## 13  0.5 0.500 0.486393207 0.0107640049
## 14  1.0 0.500 0.419518039 0.0154608300
## 15  4.0 0.500 0.394625765 0.0140754032
## 16  9.0 0.500 0.394625765 0.0140754032
```

```
tune.out$best.performance
```

```
## [1] 0.004480502
```

```
tune.out$best.model
```

```
## 
## Call:
## best.tune(method = svm, train.x = label ~ ., data = train_36, ranges = list(cost = c(0.5, 
##     1, 4, 9), gamma = c(0.001, 0.01, 0.1, 0.5)), tunecontrol = tune_ctrl, 
##     kernel = "radial", scale = TRUE)
## 
## 
## Parameters:
##    SVM-Type:  C-classification 
##  SVM-Kernel:  radial 
##        cost:  9 
## 
## Number of Support Vectors:  235
```

```
# choose cost = 4, gamma = 0.001

svmfit2=svm(label~., data=train_36, kernel="radial", gamma=0.001, cost = 4, scale=TRUE)
### prediction
ypred2=predict(svmfit2,test_36)
table(predict=ypred2, truth=test_36$label)  # confusion matrix
```

```
##        truth
## predict    3    6
##       3 1255    5
##       6    7 1195
```

```
sum(ypred2==test_36$label)/nrow(test_36)   # accuracy
```

```
## [1] 0.9951259
```

```
1-sum(ypred2==test_36$label)/nrow(test_36)  # the mis-classification error
```

```
## [1] 0.004874086
```

```
#3
# The method of radial kernel with the best parameters is a bit preciser than linear kernel. (99.51% > 99.39%)
# While the training time is much much longer than that of linear kernel.

#4

train_1258=train[train$label==1 | train$label==2 | train$label==5 | train$label==8,]
test_1258=test[test$label==1 | test$label==2 | test$label==5 | test$label==8,]

train_1258$label=as.factor(train_1258$label)
test_1258$label=as.factor(test_1258$label)

start_time=Sys.time()  # start time

tune_ctrl=tune.control(sampling = "cross", cross = 5)

set.seed(123)
tune.out=tune(svm, label~.,data=train_1258,kernel="linear",scale=TRUE,ranges=list(cost=c(0.02,0.05,0.1,0.5,1,3,8)),tunecontrol=tune_ctrl)

end_time=Sys.time()    # end time
execution_time = end_time-start_time
execution_time
```

```
## Time difference of 5.8817 mins
```

```
tune.out$performances
```

```
##   cost      error  dispersion
## 1 0.02 0.03962056 0.002309402
## 2 0.05 0.03936870 0.002417061
## 3 0.10 0.04071158 0.003065830
## 4 0.50 0.04499254 0.002703312
## 5 1.00 0.04541239 0.002321090
## 6 3.00 0.04809839 0.003994784
## 7 8.00 0.05112029 0.004430556
```

```
tune.out$best.performance
```

```
## [1] 0.0393687
```

```
tune.out$best.model
```

```
##
## Call:
## best.tune(method = svm, train.x = label ~ ., data = train_1258, ranges = list(cost = c(0.02,
##     0.05, 0.1, 0.5, 1, 3, 8)), tunecontrol = tune_ctrl, kernel = "linear",
##     scale = TRUE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.05
##
## Number of Support Vectors:  1451
```

```
# choose cost = 0.05

svmfit3=svm(label~., data=train_1258, kernel="linear", cost=0.05 , scale=TRUE)
### prediction
ypred3=predict(svmfit3,test_1258)
table(predict=ypred3, truth=test_1258$label)  # confusion matrix
```

```
##        truth
## predict    1    2    5    8
##       1 1343    5   10   17
##       2   12 1141   19   25
##       5    2   18 1063   45
##       8    6   21   34 1045
```

```
sum(ypred3==test_1258$label)/nrow(test_1258)   # accuracy
```

```
## [1] 0.9554723
```

```
1-sum(ypred3==test_1258$label)/nrow(test_1258)  # the mis-classification error
```

```
## [1] 0.04452767
```

```
#5

train$label=as.factor(train$label)
test$label=as.factor(test$label)

start_time=Sys.time()  # start time

tune_ctrl=tune.control(sampling = "cross", cross = 5)

set.seed(123)
tune.out=tune(svm, label~.,data=train,kernel="linear",scale=TRUE,ranges=list(cost=c(0.02,0.05,
0.1,0.5,2,8)),tunecontrol=tune_ctrl)

end_time=Sys.time()    # end time
execution_time = end_time-start_time
execution_time
```

```
## Time difference of 44.05747 mins
```

```
tune.out$performances
```

```
##   cost      error  dispersion
## 1 0.02 0.06250000 0.003268112
## 2 0.05 0.06223333 0.004884215
## 3 0.10 0.06250000 0.004785685
## 4 0.50 0.06576667 0.003656425
## 5 2.00 0.06926667 0.002950047
## 6 8.00 0.07200000 0.002801289
```

```
tune.out$best.performance
```

```
## [1] 0.06223333
```

```
tune.out$best.model
```

```
##
## Call:
## best.tune(method = svm, train.x = label ~ ., data = train, ranges = list(cost = c(0.02,
##     0.05, 0.1, 0.5, 2, 8)), tunecontrol = tune_ctrl, kernel = "linear",
##     scale = TRUE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.05
##
## Number of Support Vectors:  5923
```

```
# choose cost = 0.05

svmfit4=svm(label~., data=train, kernel="linear", cost=0.05 , scale=TRUE)
### prediction
ypred4=predict(svmfit4,test)
table(predict=ypred4, truth=test$label)  # confusion matrix
```

```
##        truth
## predict    0    1    2    3    4    5    6    7    8    9
##       0 1111    0    6    2    2    8    9    0    4    3
##       1    0 1340    2    6    3   12    2    3   14    5
##       2    1   10 1116   36   10    6   11   16   21    7
##       3    1    2    6 1137    1   38    0    3   22    8
##       4    1    0   21    0 1113    3    8    7    4   24
##       5    7    2    4   42    4 1015   14    5   34    6
##       6   10    1    6    4    7   18 1148    0    3    0
##       7    0    2    7    9    5    1    3 1184    3   31
##       8    7    5   14   17    1   19    4    1 1013    9
##       9    2    1    3    9   29    6    1   45   14 1060
```

```
sum(ypred4==test$label)/nrow(test)   # accuracy
```

```
## [1] 0.9364167
```

```
1-sum(ypred4==test$label)/nrow(test)  # the mis-classification error
```

```
## [1] 0.06358333
```

5.

```
#(5)
#1
import numpy as np
import pandas as pd
import random
import time

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

test=pd.read_csv("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project\\MNIST\\test_resized.csv")
train=pd.read_csv("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final project\\MNIST\\train_resized.csv")

trainy=train.loc[:,"label"].values
testy=test.loc[:,"label"].values

tent1 = train.iloc[:, 1:].values
tent1 = (tent1 - np.mean(tent1, axis=1)[:, np.newaxis]) / np.std(tent1, axis=1)[:, np.newaxis]
tent2 = test.iloc[:, 1:].values
tent2 = (tent2 - np.mean(tent2, axis=1)[:, np.newaxis]) / np.std(tent2, axis=1)[:, np.newaxis]

trainx=np.array(tent1.reshape(30000, 12, 12))
testx=np.array(tent2.reshape(12000, 12, 12))

featuresTrain = torch.from_numpy(trainx)
targetsTrain = torch.from_numpy(trainy).type(torch.LongTensor) # data type is long

featuresTest = torch.from_numpy(testx)
targetsTest = torch.from_numpy(testy).type(torch.LongTensor) # data type is long

# Pytorch train and test TensorDataset
train = torch.utils.data.TensorDataset(featuresTrain,targetsTrain)
test = torch.utils.data.TensorDataset(featuresTest,targetsTest)

###########################################################################################

# Hyper Parameters
# batch_size, epoch and iteration
LR = 0.01
batch_size = 100
n_iters = 20000
num_epochs = n_iters / (len(featuresTrain) / batch_size)
num_epochs = int(num_epochs)

# Pytorch DataLoader
train_loader = torch.utils.data.DataLoader(train, batch_size = batch_size, shuffle = True)
test_loader = torch.utils.data.DataLoader(test, batch_size = batch_size, shuffle = True)
```

```
############################################################################################

# Create CNN Model
class CNN_Model(nn.Module):
    def __init__(self):
        super(CNN_Model, self).__init__()
        # Convolution 1 , input_shape=(1,12,12)
        self.cnn1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=5, stride=1, padding=
0) #output_shape=(32,8,8)
        self.relu1 = nn.ReLU() # activation
        # Max pool 1
        self.maxpool1 = nn.MaxPool2d(kernel_size=2) #output_shape=(32,4,4)
        # Convolution 2
        self.cnn2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding
=0) #output_shape=(64,2,2)
        self.relu2 = nn.ReLU() # activation
        # Max pool 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2) #output_shape=(64,1,1)
        # Fully connected 1 ,#input_shape=(64*1*1)
        self.fc1 = nn.Linear(64 * 1 * 1, 10) #output 0-9

    def forward(self, x):
        # Convolution 1
        out = self.cnn1(x)
        out = self.relu1(out)
        # Max pool 1
        out = self.maxpool1(out)
        # Convolution 2
        out = self.cnn2(out)
        out = self.relu2(out)
        # Max pool 2
        out = self.maxpool2(out)
        out = out.view(out.size(0), -1)
        # Linear function (readout)
        out = self.fc1(out)
        return out

############################################################################################

model = CNN_Model()
print(model)
```

```
## CNN_Model(
##   (cnn1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
##   (relu1): ReLU()
##   (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
##   (cnn2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
##   (relu2): ReLU()
##   (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
##   (fc1): Linear(in_features=64, out_features=10, bias=True)
## )
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=LR)   # optimize all cnn parameters
loss_func = nn.CrossEntropyLoss()   # the target label is not one-hotted
input_shape = (-1,1,12,12)

###########################################################################################

def fit_model(model, loss_func, optimizer, input_shape, num_epochs, train_loader, test_loader):
    # Traning the Model
    #history-like list for store loss & acc value
    training_loss = []
    training_accuracy = []
    validation_loss = []
    validation_accuracy = []
    for epoch in range(num_epochs):
        #training model & store loss & acc / epoch
        correct_train = 0
        total_train = 0
        for i, (images, labels) in enumerate(train_loader):
            # 1.Define variables
            train = Variable(images.view(input_shape)).float()
            labels = Variable(labels)
            # 2.Clear gradients
            optimizer.zero_grad()
            # 3.Forward propagation
            outputs = model(train)
            # 4.Calculate softmax and cross entropy loss
            train_loss = loss_func(outputs, labels)
            # 5.Calculate gradients
            train_loss.backward()
            # 6.Update parameters
            optimizer.step()
            # 7.Get predictions from the maximum value
            predicted = torch.max(outputs.data, 1)[1]
            # 8.Total number of labels
            total_train += len(labels)
            # 9.Total correct predictions
            correct_train += (predicted == labels).float().sum()
        #10.store val_acc / epoch
        train_accuracy = 100 * correct_train / float(total_train)
        training_accuracy.append(train_accuracy)
        # 11.store loss / epoch
        training_loss.append(train_loss.data)

        #evaluate model & store loss & acc / epoch
        correct_test = 0
        total_test = 0
        for images, labels in test_loader:
            # 1.Define variables
            test = Variable(images.view(input_shape)).float()
            # 2.Forward propagation
            outputs = model(test)
            # 3.Calculate softmax and cross entropy loss
            val_loss = loss_func(outputs, labels)
            # 4.Get predictions from the maximum value
            predicted = torch.max(outputs.data, 1)[1]
```

```
            # 5.Total number of labels
            total_test += len(labels)
            # 6.Total correct predictions
            correct_test += (predicted == labels).float().sum()
        #6.store val_acc / epoch
        val_accuracy = 100 * correct_test / float(total_test)
        validation_accuracy.append(val_accuracy)
        # 11.store val_loss / epoch
        validation_loss.append(val_loss.data)
        print('Train Epoch: {}/{} Traing_Loss: {} Traing_acc: {:.6f}% Val_Loss: {} Val_accurac
y: {:.6f}%'.format(epoch+1, num_epochs, train_loss.data, train_accuracy, val_loss.data, val_acc
uracy))
    return training_loss, training_accuracy, validation_loss, validation_accuracy

###########################################################################################

start_time = time.time()

training_loss, training_accuracy, validation_loss, validation_accuracy = fit_model(model, loss_
func, optimizer, input_shape, num_epochs, train_loader, test_loader)
```

```
## Train Epoch: 1/66 Traing_Loss: 0.26835739612579346 Traing_acc: 91.519997% Val_Loss: 0.18846212327480316 Val_accuracy: 94.941666%
## Train Epoch: 2/66 Traing_Loss: 0.03799760341644287 Traing_acc: 96.946663% Val_Loss: 0.0806645080447197 Val_accuracy: 96.983330%
## Train Epoch: 3/66 Traing_Loss: 0.012601058930158615 Traing_acc: 97.653336% Val_Loss: 0.19024935364723206 Val_accuracy: 97.083336%
## Train Epoch: 4/66 Traing_Loss: 0.025363482534885406 Traing_acc: 97.860001% Val_Loss: 0.37105029821395874 Val_accuracy: 97.574997%
## Train Epoch: 5/66 Traing_Loss: 0.1273987889289856 Traing_acc: 97.836670% Val_Loss: 0.2299192100763321 Val_accuracy: 96.699997%
## Train Epoch: 6/66 Traing_Loss: 0.02357202209532261 Traing_acc: 98.173332% Val_Loss: 0.006181970704346895 Val_accuracy: 97.708336%
## Train Epoch: 7/66 Traing_Loss: 0.08227313309907913 Traing_acc: 98.283333% Val_Loss: 0.06712651252746582 Val_accuracy: 97.291664%
## Train Epoch: 8/66 Traing_Loss: 0.05351273715496063 Traing_acc: 98.400002% Val_Loss: 0.06111569330096245 Val_accuracy: 97.033333%
## Train Epoch: 9/66 Traing_Loss: 0.03841734305024147 Traing_acc: 98.550003% Val_Loss: 0.08158077299594879 Val_accuracy: 97.541664%
## Train Epoch: 10/66 Traing_Loss: 0.025341562926769257 Traing_acc: 98.396667% Val_Loss: 0.19524423778057098 Val_accuracy: 97.741669%
## Train Epoch: 11/66 Traing_Loss: 0.008716057986021042 Traing_acc: 98.650002% Val_Loss: 0.13800537586212158 Val_accuracy: 97.375000%
## Train Epoch: 12/66 Traing_Loss: 0.0657801404595375 Traing_acc: 98.516670% Val_Loss: 0.22696857154369354 Val_accuracy: 97.183334%
## Train Epoch: 13/66 Traing_Loss: 0.007539425510913134 Traing_acc: 98.620003% Val_Loss: 0.030646901577711105 Val_accuracy: 97.183334%
## Train Epoch: 14/66 Traing_Loss: 0.027419723570346832 Traing_acc: 98.373337% Val_Loss: 0.025135379284620285 Val_accuracy: 97.574997%
## Train Epoch: 15/66 Traing_Loss: 0.022285137325525284 Traing_acc: 98.766670% Val_Loss: 0.1899373084306717 Val_accuracy: 97.658333%
## Train Epoch: 16/66 Traing_Loss: 0.032083846628665924 Traing_acc: 98.706665% Val_Loss: 0.15934127569198608 Val_accuracy: 97.541664%
## Train Epoch: 17/66 Traing_Loss: 0.01242726482450962 Traing_acc: 98.669998% Val_Loss: 0.2550252676010132 Val_accuracy: 97.599998%
## Train Epoch: 18/66 Traing_Loss: 0.2558763027191162 Traing_acc: 98.726669% Val_Loss: 0.013362575322389603 Val_accuracy: 97.508331%
## Train Epoch: 19/66 Traing_Loss: 0.023177793249487877 Traing_acc: 98.766670% Val_Loss: 0.01183711364865303 Val_accuracy: 97.599998%
## Train Epoch: 20/66 Traing_Loss: 0.010979310609400272 Traing_acc: 98.983330% Val_Loss: 0.19839254021644592 Val_accuracy: 97.533333%
## Train Epoch: 21/66 Traing_Loss: 0.26897984743118286 Traing_acc: 98.783333% Val_Loss: 0.17049957811832428 Val_accuracy: 97.358330%
## Train Epoch: 22/66 Traing_Loss: 0.09984376281499863 Traing_acc: 98.769997% Val_Loss: 0.06100162863731384 Val_accuracy: 97.775002%
## Train Epoch: 23/66 Traing_Loss: 0.012712283991277218 Traing_acc: 98.916664% Val_Loss: 0.01538266334682703 Val_accuracy: 97.316666%
## Train Epoch: 24/66 Traing_Loss: 0.057563796639442444 Traing_acc: 98.796669% Val_Loss: 0.13530300557613373 Val_accuracy: 97.741669%
## Train Epoch: 25/66 Traing_Loss: 0.026566771790385246 Traing_acc: 99.110001% Val_Loss: 0.007238347083330154 Val_accuracy: 97.599998%
## Train Epoch: 26/66 Traing_Loss: 0.000487062701722607 Traing_acc: 99.036667% Val_Loss: 0.04754852131009102 Val_accuracy: 97.633331%
## Train Epoch: 27/66 Traing_Loss: 0.17444534599781036 Traing_acc: 98.636665% Val_Loss: 0.26782917976379395 Val_accuracy: 97.449997%
## Train Epoch: 28/66 Traing_Loss: 0.008254734799265862 Traing_acc: 98.846664% Val_Loss: 0.1245
```

```
3123182058334 Val_accuracy: 97.474998%
## Train Epoch: 29/66 Traing_Loss: 0.028483949601650238 Traing_acc: 99.293335% Val_Loss: 0.0017
275752034038305 Val_accuracy: 97.841667%
## Train Epoch: 30/66 Traing_Loss: 0.011739697307348251 Traing_acc: 99.156670% Val_Loss: 0.0141
79255813360214 Val_accuracy: 97.408333%
## Train Epoch: 31/66 Traing_Loss: 0.0024520272854715586 Traing_acc: 99.106667% Val_Loss: 0.006
436588242650032 Val_accuracy: 97.783333%
## Train Epoch: 32/66 Traing_Loss: 8.50549986353144e-06 Traing_acc: 99.063332% Val_Loss: 0.1389
5674049854279 Val_accuracy: 97.550003%
## Train Epoch: 33/66 Traing_Loss: 0.006649973802268505 Traing_acc: 99.080002% Val_Loss: 0.0943
2844817638397 Val_accuracy: 97.241669%
## Train Epoch: 34/66 Traing_Loss: 0.047929905354976654 Traing_acc: 98.976669% Val_Loss: 0.2778
361439704895 Val_accuracy: 97.800003%
## Train Epoch: 35/66 Traing_Loss: 0.01643337681889534 Traing_acc: 99.056664% Val_Loss: 0.59197
69406318665 Val_accuracy: 97.691666%
## Train Epoch: 36/66 Traing_Loss: 0.003279791446402669 Traing_acc: 99.230003% Val_Loss: 0.1676
289439201355 Val_accuracy: 97.633331%
## Train Epoch: 37/66 Traing_Loss: 0.030749347060918808 Traing_acc: 99.019997% Val_Loss: 0.7285
674214363098 Val_accuracy: 97.400002%
## Train Epoch: 38/66 Traing_Loss: 0.030358687043190002 Traing_acc: 99.120003% Val_Loss: 0.2337
9994928836823 Val_accuracy: 97.224998%
## Train Epoch: 39/66 Traing_Loss: 0.09197022765874863 Traing_acc: 99.186668% Val_Loss: 0.05310
463905334473 Val_accuracy: 97.375000%
## Train Epoch: 40/66 Traing_Loss: 0.02250530570745468 Traing_acc: 99.216667% Val_Loss: 1.20862
22171783447 Val_accuracy: 97.258331%
## Train Epoch: 41/66 Traing_Loss: 0.08936404436826706 Traing_acc: 99.029999% Val_Loss: 0.24217
456579208374 Val_accuracy: 97.625000%
## Train Epoch: 42/66 Traing_Loss: 0.04795249551534653 Traing_acc: 99.080002% Val_Loss: 0.04305
948689579964 Val_accuracy: 97.058334%
## Train Epoch: 43/66 Traing_Loss: 0.00023507399600930512 Traing_acc: 99.286667% Val_Loss: 0.94
02304291725159 Val_accuracy: 97.591667%
## Train Epoch: 44/66 Traing_Loss: 0.0656762346625328 Traing_acc: 99.266670% Val_Loss: 0.968422
7705001831 Val_accuracy: 97.625000%
## Train Epoch: 45/66 Traing_Loss: 0.13509349524974823 Traing_acc: 99.266670% Val_Loss: 0.23703
46039533615 Val_accuracy: 97.525002%
## Train Epoch: 46/66 Traing_Loss: 0.0931258499622345 Traing_acc: 99.276665% Val_Loss: 0.182372
00379371643 Val_accuracy: 97.858330%
## Train Epoch: 47/66 Traing_Loss: 0.039196763187646866 Traing_acc: 99.099998% Val_Loss: 0.0011
002643732354045 Val_accuracy: 97.483330%
## Train Epoch: 48/66 Traing_Loss: 3.355223725520773e-06 Traing_acc: 99.213333% Val_Loss: 0.185
98422408103943 Val_accuracy: 97.866669%
## Train Epoch: 49/66 Traing_Loss: 4.50523339168285e-06 Traing_acc: 99.446663% Val_Loss: 0.3839
084506034851 Val_accuracy: 97.608330%
## Train Epoch: 50/66 Traing_Loss: 0.029315821826457977 Traing_acc: 99.176666% Val_Loss: 6.0796
65837432913e-08 Val_accuracy: 97.441666%
## Train Epoch: 51/66 Traing_Loss: 0.122636578977108 Traing_acc: 99.336670% Val_Loss: 0.5312579
274177551 Val_accuracy: 97.500000%
## Train Epoch: 52/66 Traing_Loss: 0.00019646035798359662 Traing_acc: 99.423332% Val_Loss: 0.42
392289638519287 Val_accuracy: 97.616669%
## Train Epoch: 53/66 Traing_Loss: 0.03544505685567856 Traing_acc: 99.273331% Val_Loss: 0.01453
2673172652721 Val_accuracy: 97.158333%
## Train Epoch: 54/66 Traing_Loss: 0.06683409959077835 Traing_acc: 99.303337% Val_Loss: 0.60919
75569725037 Val_accuracy: 97.416664%
## Train Epoch: 55/66 Traing_Loss: 0.21645088493824005 Traing_acc: 99.246666% Val_Loss: 0.14232
087135314941 Val_accuracy: 96.983330%
## Train Epoch: 56/66 Traing_Loss: 0.00018999268650077283 Traing_acc: 99.333336% Val_Loss: 0.09
```

```
483062475919724 Val_accuracy: 97.574997%
## Train Epoch: 57/66 Traing_Loss: 0.0033052668441087008 Traing_acc: 99.326668% Val_Loss: 0.106
03413730859756 Val_accuracy: 97.683334%
## Train Epoch: 58/66 Traing_Loss: 0.038259051740169525 Traing_acc: 99.506668% Val_Loss: 0.6757
61342048645 Val_accuracy: 97.558334%
## Train Epoch: 59/66 Traing_Loss: 0.10571465641260147 Traing_acc: 99.496666% Val_Loss: 3.18285
2879035636e-07 Val_accuracy: 97.824997%
## Train Epoch: 60/66 Traing_Loss: 0.0006334431236609817 Traing_acc: 99.300003% Val_Loss: 0.004
058730788528919 Val_accuracy: 97.574997%
## Train Epoch: 61/66 Traing_Loss: 0.0011370916618034244 Traing_acc: 99.470001% Val_Loss: 0.080
7742029428482 Val_accuracy: 97.516670%
## Train Epoch: 62/66 Traing_Loss: 0.014317493885755539 Traing_acc: 99.500000% Val_Loss: 1.1938
966512680054 Val_accuracy: 97.191666%
## Train Epoch: 63/66 Traing_Loss: 0.0003460788866505027 Traing_acc: 99.406670% Val_Loss: 0.660
5290174484253 Val_accuracy: 97.658333%
## Train Epoch: 64/66 Traing_Loss: 0.00037960114423185587 Traing_acc: 99.459999% Val_Loss: 0.24
798643589019775 Val_accuracy: 97.491669%
## Train Epoch: 65/66 Traing_Loss: 1.2504273172453395e-06 Traing_acc: 99.273331% Val_Loss: 0.22
926448285579681 Val_accuracy: 97.408333%
## Train Epoch: 66/66 Traing_Loss: 6.258870143938111e-06 Traing_acc: 99.246666% Val_Loss: 0.849
0675091743469 Val_accuracy: 97.683334%
```

```
end_time = time.time()
runtime = end_time - start_time
print("程序运行时间：", runtime, "秒")





#2
# VAE to visulization:
```

```
## 程序运行时间： 144.91736102104187 秒
```

```
import keras
```

```
## WARNING:tensorflow:From D:\app\python\Lib\site-packages\keras\src\losses.py:2976: The name t
f.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_soft
max_cross_entropy instead.
```

```
from keras import layers

from keras.datasets import mnist
import numpy as np

original_dim = 12 * 12
intermediate_dim = 64
latent_dim = 2

inputs = keras.Input(shape=(original_dim,))
```

```
## WARNING:tensorflow:From D:\app\python\Lib\site-packages\keras\src\backend.py:1398: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.executing_eagerly_outside_functions instead.
```

```
h = layers.Dense(intermediate_dim, activation='relu')(inputs)
z_mean = layers.Dense(latent_dim)(h)
z_log_sigma = layers.Dense(latent_dim)(h)

from keras import backend as K

def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                              mean=0., stddev=0.1)
    return z_mean + K.exp(z_log_sigma) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_sigma])

# Create encoder
encoder = keras.Model(inputs, [z_mean, z_log_sigma, z], name='encoder')

# Create decoder
latent_inputs = keras.Input(shape=(latent_dim,), name='z_sampling')
x = layers.Dense(intermediate_dim, activation='relu')(latent_inputs)
outputs = layers.Dense(original_dim, activation='sigmoid')(x)
decoder = keras.Model(latent_inputs, outputs, name='decoder')

# instantiate VAE model
outputs = decoder(encoder(inputs)[2])
vae = keras.Model(inputs, outputs, name='vae_mlp')
vae.summary()
```

```
## Model: "vae_mlp"
## _________________________________________________________________
##  Layer (type)                Output Shape              Param #
## =================================================================
##  input_1 (InputLayer)        [(None, 144)]             0
##
##  encoder (Functional)        [(None, 2),               9540
##                               (None, 2),
##                               (None, 2)]
##
##  decoder (Functional)        (None, 144)               9552
##
## =================================================================
## Total params: 19092 (74.58 KB)
## Trainable params: 19092 (74.58 KB)
## Non-trainable params: 0 (0.00 Byte)
## _________________________________________________________________
```

```
reconstruction_loss = keras.losses.binary_crossentropy(inputs, outputs)
reconstruction_loss *= original_dim
kl_loss = 1 + z_log_sigma - K.square(z_mean) - K.exp(z_log_sigma)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)
vae.add_loss(vae_loss)
```

```
## WARNING:tensorflow:From D:\app\python\Lib\site-packages\keras\src\engine\base_layer_utils.p
y:384: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.e
xecuting_eagerly_outside_functions instead.
```

```
vae.compile(optimizer='adam')
```

```
## WARNING:tensorflow:From D:\app\python\Lib\site-packages\keras\src\optimizers\__init__.py:30
9: The name tf.train.Optimizer is deprecated. Please use tf.compat.v1.train.Optimizer instead.
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

def change(input_arr):

    # 缩放后每行的长度
    scaled_length = 144

    # 生成插值的位置
    interpolation_indices = np.linspace(0, input_arr.shape[1] - 1, scaled_length)

    # 进行线性插值
    scaled_arr = np.zeros((input_arr.shape[0], scaled_length))
    for i, row in enumerate(input_arr):
        scaled_arr[i] = np.interp(interpolation_indices, np.arange(row.shape[0]), row)

    # 输出结果
    return scaled_arr

train=pd.read_csv("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final proj
ect\\MNIST\\train_resized.csv")
test=pd.read_csv("C:\\Users\\张铭韬\\Desktop\\学业\\港科大\\MSDM5054机器学习\\作业\\Final proje
ct\\MNIST\\test_resized.csv")
trainy=train.loc[:,"label"].values
testy=test.loc[:,"label"].values
tent1 = train.iloc[:, 1:].values
tent1 = (tent1 - np.mean(tent1, axis=1)[:, np.newaxis]) / np.std(tent1, axis=1)[:, np.newaxis]
tent2 = test.iloc[:, 1:].values
tent2 = (tent2 - np.mean(tent2, axis=1)[:, np.newaxis]) / np.std(tent2, axis=1)[:, np.newaxis]

x1 = change(x_train)
x2 = change(x_test)
total = np.vstack((x1, x2))
totallabel = np.concatenate((y_train, y_test))

vae.fit(total, total,
        epochs=100,
        batch_size=32,
        validation_data=(total, total),
        verbose=2)
```

```
## Epoch 1/100
## 2188/2188 - 6s - loss: 34.5308 - val_loss: 30.9322 - 6s/epoch - 3ms/step
## Epoch 2/100
## 2188/2188 - 4s - loss: 30.4939 - val_loss: 30.1457 - 4s/epoch - 2ms/step
## Epoch 3/100
## 2188/2188 - 4s - loss: 29.9721 - val_loss: 29.7935 - 4s/epoch - 2ms/step
## Epoch 4/100
## 2188/2188 - 4s - loss: 29.6639 - val_loss: 29.5278 - 4s/epoch - 2ms/step
## Epoch 5/100
## 2188/2188 - 4s - loss: 29.4217 - val_loss: 29.3046 - 4s/epoch - 2ms/step
## Epoch 6/100
## 2188/2188 - 4s - loss: 29.2599 - val_loss: 29.1784 - 4s/epoch - 2ms/step
## Epoch 7/100
## 2188/2188 - 4s - loss: 29.1319 - val_loss: 29.0647 - 4s/epoch - 2ms/step
## Epoch 8/100
## 2188/2188 - 4s - loss: 29.0238 - val_loss: 28.9754 - 4s/epoch - 2ms/step
## Epoch 9/100
## 2188/2188 - 4s - loss: 28.9391 - val_loss: 28.8647 - 4s/epoch - 2ms/step
## Epoch 10/100
## 2188/2188 - 4s - loss: 28.8538 - val_loss: 28.8073 - 4s/epoch - 2ms/step
## Epoch 11/100
## 2188/2188 - 4s - loss: 28.7872 - val_loss: 28.7593 - 4s/epoch - 2ms/step
## Epoch 12/100
## 2188/2188 - 4s - loss: 28.7223 - val_loss: 28.6763 - 4s/epoch - 2ms/step
## Epoch 13/100
## 2188/2188 - 4s - loss: 28.6670 - val_loss: 28.6152 - 4s/epoch - 2ms/step
## Epoch 14/100
## 2188/2188 - 4s - loss: 28.6106 - val_loss: 28.5482 - 4s/epoch - 2ms/step
## Epoch 15/100
## 2188/2188 - 4s - loss: 28.5677 - val_loss: 28.5145 - 4s/epoch - 2ms/step
## Epoch 16/100
## 2188/2188 - 4s - loss: 28.5271 - val_loss: 28.4713 - 4s/epoch - 2ms/step
## Epoch 17/100
## 2188/2188 - 4s - loss: 28.4887 - val_loss: 28.4643 - 4s/epoch - 2ms/step
## Epoch 18/100
## 2188/2188 - 4s - loss: 28.4471 - val_loss: 28.4362 - 4s/epoch - 2ms/step
## Epoch 19/100
## 2188/2188 - 4s - loss: 28.4137 - val_loss: 28.3839 - 4s/epoch - 2ms/step
## Epoch 20/100
## 2188/2188 - 4s - loss: 28.3840 - val_loss: 28.3707 - 4s/epoch - 2ms/step
## Epoch 21/100
## 2188/2188 - 4s - loss: 28.3540 - val_loss: 28.3275 - 4s/epoch - 2ms/step
## Epoch 22/100
## 2188/2188 - 4s - loss: 28.3285 - val_loss: 28.3078 - 4s/epoch - 2ms/step
## Epoch 23/100
## 2188/2188 - 4s - loss: 28.3014 - val_loss: 28.2576 - 4s/epoch - 2ms/step
## Epoch 24/100
## 2188/2188 - 4s - loss: 28.2850 - val_loss: 28.2773 - 4s/epoch - 2ms/step
## Epoch 25/100
## 2188/2188 - 4s - loss: 28.2546 - val_loss: 28.2159 - 4s/epoch - 2ms/step
## Epoch 26/100
## 2188/2188 - 4s - loss: 28.2323 - val_loss: 28.1920 - 4s/epoch - 2ms/step
## Epoch 27/100
## 2188/2188 - 4s - loss: 28.2218 - val_loss: 28.1670 - 4s/epoch - 2ms/step
## Epoch 28/100
```

```
## 2188/2188 - 4s - loss: 28.2045 - val_loss: 28.1537 - 4s/epoch - 2ms/step
## Epoch 29/100
## 2188/2188 - 4s - loss: 28.1845 - val_loss: 28.1589 - 4s/epoch - 2ms/step
## Epoch 30/100
## 2188/2188 - 4s - loss: 28.1836 - val_loss: 28.1170 - 4s/epoch - 2ms/step
## Epoch 31/100
## 2188/2188 - 4s - loss: 28.1550 - val_loss: 28.1518 - 4s/epoch - 2ms/step
## Epoch 32/100
## 2188/2188 - 4s - loss: 28.1489 - val_loss: 28.0950 - 4s/epoch - 2ms/step
## Epoch 33/100
## 2188/2188 - 4s - loss: 28.1301 - val_loss: 28.1043 - 4s/epoch - 2ms/step
## Epoch 34/100
## 2188/2188 - 5s - loss: 28.1123 - val_loss: 28.1210 - 5s/epoch - 2ms/step
## Epoch 35/100
## 2188/2188 - 4s - loss: 28.0983 - val_loss: 28.0771 - 4s/epoch - 2ms/step
## Epoch 36/100
## 2188/2188 - 4s - loss: 28.0916 - val_loss: 28.0342 - 4s/epoch - 2ms/step
## Epoch 37/100
## 2188/2188 - 4s - loss: 28.0731 - val_loss: 28.0397 - 4s/epoch - 2ms/step
## Epoch 38/100
## 2188/2188 - 4s - loss: 28.0649 - val_loss: 28.0416 - 4s/epoch - 2ms/step
## Epoch 39/100
## 2188/2188 - 4s - loss: 28.0536 - val_loss: 28.0395 - 4s/epoch - 2ms/step
## Epoch 40/100
## 2188/2188 - 4s - loss: 28.0465 - val_loss: 28.2344 - 4s/epoch - 2ms/step
## Epoch 41/100
## 2188/2188 - 4s - loss: 28.0297 - val_loss: 27.9941 - 4s/epoch - 2ms/step
## Epoch 42/100
## 2188/2188 - 4s - loss: 28.0082 - val_loss: 27.9771 - 4s/epoch - 2ms/step
## Epoch 43/100
## 2188/2188 - 4s - loss: 27.9984 - val_loss: 27.9368 - 4s/epoch - 2ms/step
## Epoch 44/100
## 2188/2188 - 4s - loss: 27.9797 - val_loss: 27.9370 - 4s/epoch - 2ms/step
## Epoch 45/100
## 2188/2188 - 4s - loss: 27.9660 - val_loss: 27.9184 - 4s/epoch - 2ms/step
## Epoch 46/100
## 2188/2188 - 4s - loss: 27.9554 - val_loss: 27.9298 - 4s/epoch - 2ms/step
## Epoch 47/100
## 2188/2188 - 4s - loss: 27.9403 - val_loss: 27.9499 - 4s/epoch - 2ms/step
## Epoch 48/100
## 2188/2188 - 4s - loss: 27.9280 - val_loss: 27.9110 - 4s/epoch - 2ms/step
## Epoch 49/100
## 2188/2188 - 4s - loss: 27.9145 - val_loss: 27.9929 - 4s/epoch - 2ms/step
## Epoch 50/100
## 2188/2188 - 4s - loss: 27.9120 - val_loss: 27.9022 - 4s/epoch - 2ms/step
## Epoch 51/100
## 2188/2188 - 4s - loss: 27.8947 - val_loss: 27.8550 - 4s/epoch - 2ms/step
## Epoch 52/100
## 2188/2188 - 4s - loss: 27.8858 - val_loss: 27.8963 - 4s/epoch - 2ms/step
## Epoch 53/100
## 2188/2188 - 4s - loss: 27.8828 - val_loss: 27.8679 - 4s/epoch - 2ms/step
## Epoch 54/100
## 2188/2188 - 4s - loss: 27.8713 - val_loss: 27.7817 - 4s/epoch - 2ms/step
## Epoch 55/100
## 2188/2188 - 4s - loss: 27.8533 - val_loss: 27.8104 - 4s/epoch - 2ms/step
## Epoch 56/100
```

```
## 2188/2188 - 4s - loss: 27.8446 - val_loss: 27.8092 - 4s/epoch - 2ms/step
## Epoch 57/100
## 2188/2188 - 4s - loss: 27.8369 - val_loss: 27.8419 - 4s/epoch - 2ms/step
## Epoch 58/100
## 2188/2188 - 4s - loss: 27.8300 - val_loss: 27.7955 - 4s/epoch - 2ms/step
## Epoch 59/100
## 2188/2188 - 4s - loss: 27.8077 - val_loss: 27.8440 - 4s/epoch - 2ms/step
## Epoch 60/100
## 2188/2188 - 4s - loss: 27.8112 - val_loss: 27.7634 - 4s/epoch - 2ms/step
## Epoch 61/100
## 2188/2188 - 4s - loss: 27.7878 - val_loss: 27.7459 - 4s/epoch - 2ms/step
## Epoch 62/100
## 2188/2188 - 4s - loss: 27.7810 - val_loss: 27.7408 - 4s/epoch - 2ms/step
## Epoch 63/100
## 2188/2188 - 4s - loss: 27.7702 - val_loss: 27.7352 - 4s/epoch - 2ms/step
## Epoch 64/100
## 2188/2188 - 4s - loss: 27.7593 - val_loss: 27.7320 - 4s/epoch - 2ms/step
## Epoch 65/100
## 2188/2188 - 4s - loss: 27.7535 - val_loss: 27.7500 - 4s/epoch - 2ms/step
## Epoch 66/100
## 2188/2188 - 4s - loss: 27.7423 - val_loss: 27.6872 - 4s/epoch - 2ms/step
## Epoch 67/100
## 2188/2188 - 4s - loss: 27.7358 - val_loss: 27.7598 - 4s/epoch - 2ms/step
## Epoch 68/100
## 2188/2188 - 4s - loss: 27.7226 - val_loss: 27.6784 - 4s/epoch - 2ms/step
## Epoch 69/100
## 2188/2188 - 4s - loss: 27.7177 - val_loss: 27.6872 - 4s/epoch - 2ms/step
## Epoch 70/100
## 2188/2188 - 4s - loss: 27.7139 - val_loss: 27.6827 - 4s/epoch - 2ms/step
## Epoch 71/100
## 2188/2188 - 4s - loss: 27.7011 - val_loss: 27.6786 - 4s/epoch - 2ms/step
## Epoch 72/100
## 2188/2188 - 4s - loss: 27.6889 - val_loss: 27.6322 - 4s/epoch - 2ms/step
## Epoch 73/100
## 2188/2188 - 4s - loss: 27.6864 - val_loss: 27.6307 - 4s/epoch - 2ms/step
## Epoch 74/100
## 2188/2188 - 4s - loss: 27.6825 - val_loss: 27.6235 - 4s/epoch - 2ms/step
## Epoch 75/100
## 2188/2188 - 4s - loss: 27.6708 - val_loss: 27.6130 - 4s/epoch - 2ms/step
## Epoch 76/100
## 2188/2188 - 4s - loss: 27.6645 - val_loss: 27.6301 - 4s/epoch - 2ms/step
## Epoch 77/100
## 2188/2188 - 4s - loss: 27.6662 - val_loss: 27.6567 - 4s/epoch - 2ms/step
## Epoch 78/100
## 2188/2188 - 4s - loss: 27.6533 - val_loss: 27.6211 - 4s/epoch - 2ms/step
## Epoch 79/100
## 2188/2188 - 4s - loss: 27.6476 - val_loss: 27.6007 - 4s/epoch - 2ms/step
## Epoch 80/100
## 2188/2188 - 4s - loss: 27.6392 - val_loss: 27.5730 - 4s/epoch - 2ms/step
## Epoch 81/100
## 2188/2188 - 4s - loss: 27.6299 - val_loss: 27.6033 - 4s/epoch - 2ms/step
## Epoch 82/100
## 2188/2188 - 4s - loss: 27.6371 - val_loss: 27.5696 - 4s/epoch - 2ms/step
## Epoch 83/100
## 2188/2188 - 4s - loss: 27.6234 - val_loss: 27.5649 - 4s/epoch - 2ms/step
## Epoch 84/100
```

```
## 2188/2188 - 4s - loss: 27.6259 - val_loss: 27.6249 - 4s/epoch - 2ms/step
## Epoch 85/100
## 2188/2188 - 4s - loss: 27.6104 - val_loss: 27.5729 - 4s/epoch - 2ms/step
## Epoch 86/100
## 2188/2188 - 4s - loss: 27.6121 - val_loss: 27.5984 - 4s/epoch - 2ms/step
## Epoch 87/100
## 2188/2188 - 4s - loss: 27.6001 - val_loss: 27.6317 - 4s/epoch - 2ms/step
## Epoch 88/100
## 2188/2188 - 4s - loss: 27.5964 - val_loss: 27.5727 - 4s/epoch - 2ms/step
## Epoch 89/100
## 2188/2188 - 4s - loss: 27.5949 - val_loss: 27.5392 - 4s/epoch - 2ms/step
## Epoch 90/100
## 2188/2188 - 4s - loss: 27.5829 - val_loss: 27.5992 - 4s/epoch - 2ms/step
## Epoch 91/100
## 2188/2188 - 4s - loss: 27.5800 - val_loss: 27.5386 - 4s/epoch - 2ms/step
## Epoch 92/100
## 2188/2188 - 4s - loss: 27.5771 - val_loss: 27.5053 - 4s/epoch - 2ms/step
## Epoch 93/100
## 2188/2188 - 4s - loss: 27.5709 - val_loss: 27.6011 - 4s/epoch - 2ms/step
## Epoch 94/100
## 2188/2188 - 4s - loss: 27.5706 - val_loss: 27.5469 - 4s/epoch - 2ms/step
## Epoch 95/100
## 2188/2188 - 4s - loss: 27.5632 - val_loss: 27.4965 - 4s/epoch - 2ms/step
## Epoch 96/100
## 2188/2188 - 4s - loss: 27.5651 - val_loss: 27.4895 - 4s/epoch - 2ms/step
## Epoch 97/100
## 2188/2188 - 4s - loss: 27.5497 - val_loss: 27.5238 - 4s/epoch - 2ms/step
## Epoch 98/100
## 2188/2188 - 4s - loss: 27.5456 - val_loss: 27.5189 - 4s/epoch - 2ms/step
## Epoch 99/100
## 2188/2188 - 4s - loss: 27.5418 - val_loss: 27.5173 - 4s/epoch - 2ms/step
## Epoch 100/100
## 2188/2188 - 4s - loss: 27.5441 - val_loss: 27.5420 - 4s/epoch - 2ms/step
## <keras.src.callbacks.History object at 0x00000002F2F2E0D0>
```

```
x_test_encoded = encoder.predict(total)
```

```
##
##    1/2188 [..............................] - ETA: 3:09
##   60/2188 [..............................] - ETA: 1s
##  119/2188 [>.............................] - ETA: 1s
##  172/2188 [=>............................] - ETA: 1s
##  230/2188 [==>...........................] - ETA: 1s
##  288/2188 [==>...........................] - ETA: 1s
##  346/2188 [===>..........................] - ETA: 1s
##  404/2188 [====>.........................] - ETA: 1s
##  462/2188 [=====>........................] - ETA: 1s
##  518/2188 [======>.......................] - ETA: 1s
##  577/2188 [======>.......................] - ETA: 1s
##  635/2188 [=======>......................] - ETA: 1s
##  694/2188 [========>.....................] - ETA: 1s
##  751/2188 [=========>....................] - ETA: 1s
##  811/2188 [==========>...................] - ETA: 1s
##  870/2188 [==========>...................] - ETA: 1s
##  927/2188 [===========>..................] - ETA: 1s
##  986/2188 [============>.................] - ETA: 1s
## 1042/2188 [=============>................] - ETA: 1s
## 1095/2188 [==============>...............] - ETA: 0s
## 1150/2188 [==============>...............] - ETA: 0s
## 1207/2188 [===============>..............] - ETA: 0s
## 1265/2188 [================>.............] - ETA: 0s
## 1323/2188 [=================>............] - ETA: 0s
## 1381/2188 [=================>............] - ETA: 0s
## 1438/2188 [==================>...........] - ETA: 0s
## 1496/2188 [===================>..........] - ETA: 0s
## 1554/2188 [====================>.........] - ETA: 0s
## 1612/2188 [=====================>........] - ETA: 0s
## 1670/2188 [=====================>........] - ETA: 0s
## 1728/2188 [======================>.......] - ETA: 0s
## 1782/2188 [=======================>......] - ETA: 0s
## 1840/2188 [========================>.....] - ETA: 0s
## 1897/2188 [=========================>....] - ETA: 0s
## 1954/2188 [=========================>....] - ETA: 0s
## 2013/2188 [==========================>...] - ETA: 0s
## 2073/2188 [===========================>..] - ETA: 0s
## 2132/2188 [============================>.] - ETA: 0s
## 2188/2188 [==============================] - 2s 877us/step
```

```
plt.figure(figsize=(16, 16))
plt.scatter(x_test_encoded[0][:, 0], x_test_encoded[0][:, 1], c=totallabel, cmap='Set1',s=6)
plt.colorbar()
```

```
## <matplotlib.colorbar.Colorbar object at 0x00000002DE5F3B90>
```

```
plt.show()

##########################################################################################
#######                        CAE example:

# import keras
# from keras import layers
#
# input_img = keras.Input(shape=(28, 28, 1))
#
# x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
# x = layers.MaxPooling2D((2, 2), padding='same')(x)
# x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
# x = layers.MaxPooling2D((2, 2), padding='same')(x)
# x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
# encoded = layers.MaxPooling2D((2, 2), padding='same')(x)
#
# # at this point the representation is (4, 4, 8) i.e. 128-dimensional
#
# x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
# x = layers.UpSampling2D((2, 2))(x)
# x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
# x = layers.UpSampling2D((2, 2))(x)
# x = layers.Conv2D(16, (3, 3), activation='relu')(x)
# x = layers.UpSampling2D((2, 2))(x)
# decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
#
# autoencoder = keras.Model(input_img, decoded)
# autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
#
# from keras.datasets import mnist
# import numpy as np
#
# (x_train, _), (x_test, _) = mnist.load_data()
#
# x_train = x_train.astype('float32') / 255.
# x_test = x_test.astype('float32') / 255.
# x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
# x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
#
# from keras.callbacks import TensorBoard
#
# autoencoder.fit(x_train, x_train,
#                 epochs=50,
#                 batch_size=128,
#                 shuffle=True,
#                 validation_data=(x_test, x_test),
#                 verbose=2,
#                 callbacks=[TensorBoard(log_dir='/tmp/autoencoder')])
#
# decoded_imgs = autoencoder.predict(x_test)
#
# n = 10
# plt.figure(figsize=(20, 4))
# for i in range(1, n + 1):
```

```
#     # Display original
#     ax = plt.subplot(2, n, i)
#     plt.imshow(x_test[i+100].reshape(28, 28))
#     plt.gray()
#     ax.get_xaxis().set_visible(False)
#     ax.get_yaxis().set_visible(False)
#
#     # Display reconstruction
#     ax = plt.subplot(2, n, i + n)
#     plt.imshow(decoded_imgs[i++100].reshape(28, 28))
#     plt.gray()
#     ax.get_xaxis().set_visible(False)
#     ax.get_yaxis().set_visible(False)
# plt.show()
```