Yonghyun_Lab4

April 16, 2020

[1]:
```
import numpy as np
import pandas as pd

import sklearn
from sklearn.datasets import make_classification

from sklearn.metrics import confusion_matrix

from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier,␣
↪VotingClassifier

from sklearn.ensemble import StackingClassifier

from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier

print("Numpy version = %s" % np.__version__)
print("Pandas version = %s" % pd.__version__)
print("Sklearn version = %s" % sklearn.__version__)
```

```
Numpy version = 1.18.1
Pandas version = 1.0.3
Sklearn version = 0.22.2.post1
```

1 Data import

[2]:
```
train = pd.read_csv("lab4-train.csv")
train = np.array(train)
test = pd.read_csv("lab4-test.csv")
test = np.array(test)
```

[3]:
```
trainX, trainY = np.hsplit(train, np.array([4]))
testX, testY = np.hsplit(test, np.array([4]))
trainY = trainY.flatten()
```

```
testY = testY.flatten()
```

2 Tasks 1

2.1 Training Random Forest

[4]:
```
print("--------Random Forest--------")
n_estimators = [10, 50, 100]
max_depth = [5, None]
min_samples_split = [2, 5, 10]
min_samples_leaf = [1, 5, 10]
criterion= ["gini", "entropy"]
res = [[RandomForestClassifier(n_estimators=i, max_depth=j, random_state=0,␣
 ↪min_samples_split = k,
                              min_samples_leaf = l, criterion = m).fit(trainX,␣
 ↪trainY).score(testX, testY),
        i, j, k, l, m] for i in n_estimators
                 for j in max_depth
                 for k in min_samples_split
                 for l in min_samples_leaf
                 for m in criterion]
res = pd.DataFrame(res)
res.columns = ["acc", "n_estimators", "max_depth", "min_split", "min_leaf",␣
 ↪"criterion"]
print(res)
idx = res['acc'].idxmax()
residx = res.iloc[idx]
```

```
--------Random Forest--------
          acc  n_estimators  max_depth  min_split  min_leaf criterion
0    0.820598            10        5.0          2         1      gini
1    0.823920            10        5.0          2         1   entropy
2    0.837209            10        5.0          2         5      gini
3    0.823920            10        5.0          2         5   entropy
4    0.827243            10        5.0          2        10      gini
..        …             …         …          …         …
103  0.803987           100        NaN         10         1   entropy
104  0.830565           100        NaN         10         5      gini
105  0.827243           100        NaN         10         5   entropy
106  0.830565           100        NaN         10        10      gini
107  0.833887           100        NaN         10        10   entropy

[108 rows x 6 columns]
```

[5]:
```
print("--------Random Forest--------")
clf1 = RandomForestClassifier(n_estimators=residx["n_estimators"],␣
 ↪max_depth=residx["max_depth"], random_state=0,
                              min_samples_split = residx["min_split"],
                              min_samples_leaf = residx["min_leaf"], criterion␣
 ↪= residx["criterion"])
clf1.fit(trainX, trainY)

print("Hyper-parameters for the best model:")
print(res.iloc[[idx]]); print("")

acc1test = clf1.score(testX, testY)
acc1train = clf1.score(trainX, trainY)
print("classification accuracy of test data= %g" % acc1test)
print("classification accuracy of training data= %g" % acc1train)

print("Confusion matrix for test data is")
mat = confusion_matrix(testY, clf1.predict(testX))
print(mat) # confusion matrix
print("Confusion matrix for training data is")
mat = confusion_matrix(trainY, clf1.predict(trainX))
print(mat) # confusion matrix
print("")
```

```
--------Random Forest--------
Hyper-parameters for the best model:
         acc  n_estimators  max_depth  min_split  min_leaf criterion
84  0.847176           100        5.0         10         1      gini

classification accuracy of test data= 0.847176
classification accuracy of training data= 0.803132
Confusion matrix for test data is
[[228  10]
 [ 36  27]]
Confusion matrix for training data is
[[314  18]
 [ 70  45]]
```

2.2 Training AdaBoost.M1

[6]:
```
print("--------AdaBoost.M1--------")
n_estimators = [10, 50, 100, 150]
learning_rate = [0.5, 1, 1.5]
res = [[AdaBoostClassifier(n_estimators=i, learning_rate=j, random_state=0).
 ↪fit(trainX, trainY).score(testX, testY),
```

```
        i, j] for i in n_estimators
              for j in learning_rate]
res = pd.DataFrame(res)
res.columns = ["acc", "n_estimators", "learning_rate"]
print(res)
idx = res['acc'].idxmax()
residx = res.iloc[idx]
```

```
--------AdaBoost.M1--------
         acc  n_estimators  learning_rate
0   0.797342            10            0.5
1   0.823920            10            1.0
2   0.794020            10            1.5
3   0.803987            50            0.5
4   0.810631            50            1.0
5   0.774086            50            1.5
6   0.810631           100            0.5
7   0.803987           100            1.0
8   0.780731           100            1.5
9   0.807309           150            0.5
10  0.787375           150            1.0
11  0.777409           150            1.5
```

[7]:
```
print("--------AdaBoost.M1--------")
clf2 = AdaBoostClassifier(n_estimators=(int) (residx["n_estimators"]),␣
 ↪learning_rate=residx["learning_rate"], random_state=0)
clf2.fit(trainX, trainY)

print("Hyper-parameters for the best model:")
print(res.iloc[[idx]]); print("")

acc2test = clf2.score(testX, testY)
acc2train = clf2.score(trainX, trainY)
print("classification accuracy of test data= %g" % acc2test)
print("classification accuracy of training data= %g" % acc2train)

print("Confusion matrix for test data is")
mat = confusion_matrix(testY, clf2.predict(testX))
print(mat) # confusion matrix
print("Confusion matrix for training data is")
mat = confusion_matrix(trainY, clf2.predict(trainX))
print(mat) # confusion matrix
print("")
```

```
--------AdaBoost.M1--------
Hyper-parameters for the best model:
       acc  n_estimators  learning_rate
```

```
1  0.82392            10            1.0

classification accuracy of test data= 0.82392
classification accuracy of training data= 0.787472
Confusion matrix for test data is
[[230   8]
 [ 45  18]]
Confusion matrix for training data is
[[316  16]
 [ 79  36]]
```

2.3 Discussion

It turns out that both Random Forest and AdaBoost.M1 returns similar accuracies, which are acceptable(for Ramdom forest, 0.847176, while for AdaBoost.M1, 0.82392). According to the confusion matrix, a lot of data were classifeid as 0 even though the true class is 1. This shows the difficulty of classification in this specific data even we are dealing with binary classification problem. Also, while experimenting with different hyper-parameters, we could see that the training accuracy is not necessarily higher than the test accuracy. This implies both Random Forest and AdaBoost.M1 are a good way to avoid overftting.

In the later part of this report, we will see that both Random Forest and AdaBoost outperforms the method when a single model is used. This clearly shows that ensemble learning is a good way to combine different models and improve performance.

[8]:
```
print(clf1.feature_importances_)
```

```
[0.29624974 0.18063452 0.18866312 0.33445261]
```

[9]:
```
print(clf2.feature_importances_)
```

```
[0.3 0.3 0.1 0.3]
```

We can see that each weight of features are different. When using Random Forest, the second feature is equally weighted as the third feature. However, when fitting model using AdaBoost, the second feature is weighted as three times as the first feature. Even though they have different feature importance weight, they enhance accuracy of classifciation, even when it is quite difficult to do, by combining several individual decision in a descent way.

3 Task 2

3.1 Training four individual models

3.1.1 Neural Network

Training a model

[10]:
```
print("--------Neural Network--------")
clf1 = MLPClassifier(hidden_layer_sizes=(100, 2), random_state=1, max_iter =␣
↪1000)
clf1.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf1.predict(testX))
print(mat) # confusion matrix
```

```
--------Neural Network--------
Confusion matrix is
[[228  10]
 [ 47  16]]
```

[11]:
```
acc1 = clf1.score(testX, testY) # classification accuracy
print("classification accuracy = %g" % acc1)
```

```
classification accuracy = 0.810631
```

Tuning the hyper parameter Possible hyper parameters for Neural Netwrok are hidden layer sizes, activation function, batch size, learning rate, momentum, maximum number of iterations ,and random seed. We experiment with different hidden layer sizes, batch size, learning rate and momentum.

Hidden layer sizes: (100,2) -> (90, 2)

[12]:
```
print("*** Hidden layer sizes: (100,2) -> (90, 2) ***")
clf = MLPClassifier(hidden_layer_sizes=(90, 2), random_state=1, max_iter = 1000)
clf.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** Hidden layer sizes: (100,2) -> (90, 2) ***
Confusion matrix is
[[214  24]
 [ 34  29]]
```

Batch sizes: 200 -> 190

[13]:
```
print("*** Batch sizes: 200 -> 190 ***")
# Default batch size is min(200, n_samples) = 200
clf = MLPClassifier(hidden_layer_sizes=(100, 2), random_state=1, max_iter =␣
↪1000, batch_size = 190)
clf.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** Batch sizes: 200 -> 190 ***
Confusion matrix is
[[237   1]
 [ 58   5]]
```

(Intial) learning rate: 0.001 -> 0.00101

[14]:
```
print("*** (Intial) learning rate: 0.001 -> 0.00101 ***")
# Default learning_rate_init is 0.001
clf = MLPClassifier(hidden_layer_sizes=(100, 2), random_state=1, max_iter =␣
 ↪1000, learning_rate_init = 0.00101)
clf.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** (Intial) learning rate: 0.001 -> 0.00101 ***
Confusion matrix is
[[197  41]
 [ 24  39]]
```

momentum: 0.9 -> 1

[15]:
```
print("*** momentum: 0.9 -> 1 ***")
# Default momentum is 0.9
clf = MLPClassifier(hidden_layer_sizes=(100, 2), random_state=1, max_iter =␣
 ↪1000, momentum = 1)
clf.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** momentum: 0.9 -> 1 ***
Confusion matrix is
[[228  10]
 [ 47  16]]
```

Discussion We can observe that the confusion matrix is greatly different when experimenting with different values of Hddien layer size, Batch size and Learning rate. Therefore, we need to focus more on deciding the hyper-parameters of these values. When determining momentum, however, we may not be careful, because a slight change on momentum gave same confusion matrix.

3.1.2 Logistic Regression

[16]:
```
print("--------Logistic Regression--------")
clf2 = LogisticRegression(random_state=0, solver = "liblinear").fit(trainX,␣
↪trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf2.predict(testX))
print(mat) # confusion matrix
```

```
--------Logistic Regression--------
Confusion matrix is
[[232   6]
 [ 52  11]]
```

[17]:
```
acc2 = clf2.score(testX, testY) # classification accuracy
print("classification accuracy = %g" % acc2)
```

```
classification accuracy = 0.807309
```

Tuning the hyper parameter Possible hyper parameters for Logistic Regression are penaltization norm, tolerance for stopping criteria, intercept scaling, class weight, randome seed, solver, and max_iter. We experiment with different values of tolerence, and intercept scaling.

tolerance: 0.0001 -> 0.00011

[18]:
```
print("*** tolerance: 0.0001 -> 0.00011 ***")
# Default momentum is 0.0001
clf = LogisticRegression(random_state=0, solver = "liblinear", tol=0.00011).
↪fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** tolerance: 0.0001 -> 0.00011 ***
Confusion matrix is
[[232   6]
 [ 52  11]]
```

intercept_scaling: 1 -> 1.1

[19]:
```
print("*** intercept_scaling: 1 -> 1.1 ***")
# Default intercept_scaling is 1
clf = LogisticRegression(random_state=0, solver = "liblinear",␣
↪intercept_scaling=1.1).fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** intercept_scaling: 1 -> 1.1 ***
Confusion matrix is
[[232   6]
 [ 52  11]]
```

Discussion When stlightly tuning the hyper-parameters(tolerance and intercep_scaling), none of them had great impact on the confusion matrix. This implies that a model fitted by Logistic Regression does not vary much with different values of hyperparameters.

3.1.3 Naive Bayes

[20]:
```python
print("--------Naive Bayes--------")
clf3 = GaussianNB()
clf3.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf3.predict(testX))
print(mat) # confusion matrix
```

```
--------Naive Bayes--------
Confusion matrix is
[[226  12]
 [ 48  15]]
```

[21]:
```python
acc3 = clf3.score(testX, testY) # classification accuracy
print("classification accuracy = %g" % acc3)
```

```
classification accuracy = 0.800664
```

Tuning the hyper parameter Possible hyper parameters for Naive Bayes are prior probabilities of the classes and var_smoothing. We experiment with different values of these hyperparameters.

class prior : [0.742729, 0.257271] -> [0.75, 0.25]

[22]:
```python
print("class prior = [%g, %g]" % tuple(clf3.class_prior_))
```

```
class prior = [0.742729, 0.257271]
```

[23]:
```python
print("*** class prior : [0.742729, 0.257271] -> [0.75, 0.25] ***")
clf = GaussianNB(priors = np.array([0.75, 0.25]))
clf.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** class prior : [0.742729, 0.257271] -> [0.75, 0.25] ***
Confusion matrix is
```

```
[[226  12]
 [ 48  15]]
```

var_smoothing: 1e-09 -> 1e-08

[24]:
```
print("*** var_smoothing: 1e-09 -> 1e-08 ***")
# Default momentum is 1e-09
clf = GaussianNB(var_smoothing=1e-08)
clf.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** var_smoothing: 1e-09 -> 1e-08 ***
Confusion matrix is
[[226  12]
 [ 48  15]]
```

Discussion We can see that there is no big difference when changing the hyper-parameters of Naive Bayes. Therefore, Naive Bayes is also stable in that the perturbation on hyperparameters does not result in the big difference in accuracy.

3.1.4 Decision Tree

[25]:
```
print("--------Decision Tree--------")
clf4 = DecisionTreeClassifier(random_state=0)
clf4.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf4.predict(testX))
print(mat) # confusion matrix
```

```
--------Decision Tree--------
Confusion matrix is
[[196  42]
 [ 45  18]]
```

[26]:
```
acc4 = clf4.score(testX, testY) # classification accuracy
print("classification accuracy = %g" % acc4)
```

```
classification accuracy = 0.710963
```

Tuning the hyper parameter Possible hyper parameters for Naive Bayes are criterion for split, maximum depth of a tree and the minimum number of samples required to split an internal node. We experiment with different maximum depth of a tree.

max_depth = 9

[27]:
```
print("*** class prior : [0.742729, 0.257271] -> [0.75, 0.25] ***")
clf = DecisionTreeClassifier(random_state=0, max_depth = 9)
clf.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** class prior : [0.742729, 0.257271] -> [0.75, 0.25] ***
Confusion matrix is
[[199  39]
 [ 44  19]]
```

max_depth = 10

[28]:
```
print("*** class prior : [0.742729, 0.257271] -> [0.75, 0.25] ***")
clf = DecisionTreeClassifier(random_state=0, max_depth = 10)
clf.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, clf.predict(testX))
print(mat) # confusion matrix
```

```
*** class prior : [0.742729, 0.257271] -> [0.75, 0.25] ***
Confusion matrix is
[[198  40]
 [ 45  18]]
```

Discussion We can see that there is no big difference when changing the maximum depth of a tree. Therefore, Decision Tree does not give different results based on slight modification on hyper-parameter max_depth.

3.2 Ensemble classifier using unweighted majority vote

[29]:
```
estimators = [('NN', clf1), ('LR', clf2), ('NB', clf3), ('DT', clf4)]

print("--------Unweighted majority vote--------")
eclf1 = VotingClassifier(estimators=estimators, voting='hard')
eclf1 = eclf1.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, eclf1.predict(testX))
print(mat) # confusion matrix
```

```
--------Unweighted majority vote--------
Confusion matrix is
[[232   6]
 [ 52  11]]
```

[30]:
```
acc = eclf1.score(testX, testY) # classification accuracy
print("classification accuracy = %g" % acc)
```

```
classification accuracy = 0.807309
```

The performance on the test data is just as same as that for using Logistic Regression only.(= 0.807309)

3.3 Ensiemble classifier using weighted majority vote

3.3.1 Weights proportional to the classification accuracy

[31]:
```
weights = np.array([acc1, acc2, acc3, acc4])

print("--------Weighted majority vote--------")
print("--------Weights proportional to the classification accuracy--------")
eclf2 = VotingClassifier(estimators=estimators, voting='hard', weights =␣
 ↪weights)
eclf2 = eclf2.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, eclf2.predict(testX))
print(mat) # confusion matrix
```

```
--------Weighted majority vote--------
--------Weights proportional to the classification accuracy--------
Confusion matrix is
[[230   8]
 [ 50  13]]
```

[32]:
```
acc = eclf2.score(testX, testY) # classification accuracy
print("classification accuracy = %g" % acc)
```

```
classification accuracy = 0.807309
```

3.3.2 Stacking

One can make use of *StackingClassifier* in *sklearn.ensemble* package to use stacking

[33]:
```
print("--------Stacking--------")
eclf3 = StackingClassifier(estimators=estimators, final_estimator=clf2)
eclf3 = eclf3.fit(trainX, trainY)
print("Confusion matrix is")
mat = confusion_matrix(testY, eclf3.predict(testX))
print(mat) # confusion matrix
```

```
--------Stacking--------
Confusion matrix is
```

```
[[231   7]
 [ 55   8]]
```

[34]:
```
acc = eclf3.score(testX, testY) # classification accuracy
print("classification accuracy = %g" % acc)
```

```
classification accuracy = 0.79402
```

3.4 Discussion

We performed ensemble learnings based on three different method.

1) Unweighted majority vote
2) Weighted majority vote using weights proportional to the classification accuracy
3) Stacking

It turns out that first and second methods gives the same calssification accuracy(0.807309) while Stacking perform slightly poorly(0.79402). Weighted and unweighted majority votes seems to bring similar results because the classification accuracies of four models(NN = 0.810631, LR = 0.807309, NB = 0.800664, DT = 0.710963) are similar except DT.

Since it turned out that the performance of Neural Network varies a lot according to different values of hyperparameters, one may try to improve the performance of Neural Network using ensemble learning such as Bagging or Boosting.

Even though it is impressive that the unweighted ensemble learning performs as better as weighted ensemble learning, it is possilbe that weighted learning outperforms unweighted case when the performance of each learning algorithm varies significantly.

Also, note that classifciation accuracy of Neural network is higher than that of ensemble learning we performed. Therefore, we should not blindly think that ensemble learning always give better accuracy and also have to try other methods to enhance the performance of ensemble learning.