机器学习实验报告 (五)

2018/12/26

目录

1 实验环境

操作系统	win10
编程语言	python3
编程环境	Jupyter Notebook
报告编写	latex

2 问题描述与分析

问题描述：使用 *knn*、*svm*、贝叶斯等模型，使用投票的方式对 *MNIST* 数据集训练，完成一个集成学习。

2.1 集成学习

集成学习是使用一系列学习器进行学习，并使用某种规则把各个学习结果进行整合从而获得比单个学习器更好的学习效果的一种机器学习方法。

一般情况下，有两种选择。第一种就是所有的个体学习器都是一个种类的，或者说是同质的。比如都是决策树个体学习器，或者都是神经网络个体学习器。第二种是所有的个体学习器不全是一个种类的，或者说是异质的。比如我们有一个分类问题，对训练集采用支持向量机个体学习器，逻辑回归个体学习器和朴素贝叶斯个体学习器来学习，再通过某种结合策略来确定最终的分类强学习器。

此次试验中我们将不同种类个体学习器结合在一起，观察效果。

2.2 支持向量机

支持向量机的基本模型是在特征空间上找到最佳的分离超平面使得训练集上正负样本间隔最大，一般有三种形式：硬间隔支持向量机、软间隔支持向量机、非线性支持向量机。其中非线性支持向量机采用核方法，不同的核方法所购造出来的支持向量机得性能也有很大的差别。

本次实验使用 LIBSVM（台湾大学林智仁教授等开发设计的一个简单、易于使用和快速有效的 SVM 模式识别与回归的软件包）进行实验

2.3 神经网络

BP 神经网络，是 1986 年由 Rumelhart 和 McClelland 为首的科学家提出的概念，是一种按照误差逆向传播算法训练的多层前馈神经网络。BP 算法是一种有效的多层神经网络学习方法，通过向信号前向输出层传递，误差向输入层反馈，不断调节网络权重值，使得网络的最终输出与期望输出尽可能接近，以达到训练的目的。一般来说，一个多层神经网络由输入层，若干隐含层，输出层构成。

本次试验通过在上次实验中实现的神经网络来进行实验

2.4 K 近邻

KNN 是通过测量不同特征值之间的距离进行分类。主要思路为: 如果一个样本在特征空间中的 k 个最相似的样本中的大多数属于某一个类别，则该样本也属于这个类别，其中 K 通常是不大于 20 的整数。

在 KNN 中，通过计算对象间距离来作为各个对象之间的非相似性指标，避免了对象之间的匹配问题同时，KNN 通过依据 k 个对象中占优的类别进行决策，而不是单一的对象类别决策。这两点就是 KNN 算法的优势。

KNN 算法的结果很大程度取决于 K 的选择与距离的度量。因此此次实验选择调用 sklearn 库进行实验

2.5 朴素贝叶斯

贝叶斯分类是一类分类算法的总称，这类算法均以贝叶斯定理为基础，故统称为贝叶斯分类。而朴素朴素贝叶斯分类是贝叶斯分类中最简单，也是常见的一种分类方法。

此次实验选择调用 sklearn 库进行实验

3 具体过程

3.1 MNIST 数据集的导入

在导入所需要的库依赖之后，我们首先要做的就是讲 MNIST 数据集读入。通过参考网上相关资料，我在这定义了 $load\_mnist\_train(path)$ 和 $load\_mnist\_test(path)$ 两个方法，分别用于加载训练集与测试集

```
# 加载数据
def load_mnist_train(path):
    labels_path = os.path.join(path,'train-labels.idx1-ubyte')
    images_path = os.path.join(path, 'train-images.idx3-ubyte')
    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',lbpath.read(8))
        labels = np.fromfile(lbpath, dtype=np.uint8)
    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack('>IIII',
        imgpath.read(16))
        images = np.fromfile(imgpath,
        dtype=np.uint8).reshape(len(labels), 784)
    return images, labels

def load_mnist_test(path):
    labels_path = os.path.join(path,'t10k-labels.idx1-ubyte')
    images_path = os.path.join(path, 't10k-images.idx3-ubyte')
    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',lbpath.read(8))
        labels = np.fromfile(lbpath, dtype=np.uint8)
    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack('>IIII',
```

```
        imgpath.read(16))
        images = np.fromfile(imgpath,
        dtype=np.uint8).reshape(len(labels), 784)
    return images, labels

X_train,Y_train = load_mnist_train('E:\学习\大三\机器学习\MNIST')
X_test,Y_test = load_mnist_test('E:\学习\大三\机器学习\MNIST')
```

3.2 支持向量机

此次实验采取 LIBSVM，核函数分别采用线性核与多项式核。首先导入相关的库与数据准备

```
import sys
path = 'D:\LIBSVM\libsvm-3.22\python'
sys.path.append(path)
from svmutil import *
from svm  import *

y_train_svm = Y_train.tolist()
x_train_svm=[]
index = range(1,785)
for i in X_train:
    x_train_svm.append(dict(zip(index, i)))

y_test_svm = Y_test.tolist()
x_test_svm=[]
index = range(1,785)
for i in X_test:
    x_test_svm.append(dict(zip(index, i)))
```

定义 $test\_svm(nums, typet)$ 方法，$nums$ 为训练样本数，$typet$ 为核函数类别，其中，0 代表线性核，1 代表多项式核

```
def test_svm(nums,typet):
    prob  = svm_problem(y_train_svm[:nums], x_train_svm[:nums])
    param = svm_parameter('-t␣%d␣-c␣4␣-m␣1024'%typet)
    model = svm_train(prob, param)
    p_label, p_acc, p_val = svm_predict(y_test_svm,
                                        x_test_svm, model)
    return p_acc[0],p_label
```

调用函数进行训练

```
# 线性核
acc_svm_liner,predict_svm_liner= test_svm(10000,0)
acc_svm_liner = acc_svm_liner/100
# 多项式核
acc_svm_polynomial,predict_svm_polynomial= test_svm(10000,1)
acc_svm_polynomial = acc_svm_polynomial/100
```

由于训练时间原因，我们此次只训练了 10000 个样本，得到线性核函数时准确率 0.9128，多项式核函数准确率 0.9577.

3.3 朴素贝叶斯

朴素贝叶斯是一个基于贝叶斯理论的分类器。它会单独考量每一唯独特征被分类的条件概率，进而综合这些概率并对其所在的特征向量做出分类预测。在 sklearn 中，提供了 3 中朴素贝叶斯分类算法：高斯朴素贝叶斯、多项式朴素贝叶斯、伯努利朴素贝叶斯。我们分别对其分别训练。

```
# 高斯朴素贝叶斯
from sklearn.naive_bayes import GaussianNB
```

```
clf_g = GaussianNB()
clf_g.fit(X_train,Y_train)
predict_bayes_g = clf_g.predict(X_test)
acc_bayes_g  =sum(predict_bayes_g == Y_test)/len(Y_test)
# 伯努利朴素贝叶斯
from sklearn.naive_bayes import BernoulliNB
clf_b = BernoulliNB(alpha=2.0,binarize = 3.0,fit_prior=True)
clf_b.fit(X_train,Y_train)
predict_bayes_b = clf_b.predict(X_test)
acc_bayes_b = sum(predict_bayes_b == Y_test)/len(Y_test)
# 多项式朴素贝叶斯
from sklearn.naive_bayes import MultinomialNB
clf_m = MultinomialNB(alpha=2.0)
clf_m.fit(X_train,Y_train)
predict_bayes_m = clf_m.predict(X_test)
acc_bayes_m =sum(predict_bayes_m == Y_test)/len(Y_test)
```

得到准确率分别为 0.5558、0.8423、0.8364

3.4 神经网络

在这里使用上次实验实现的多层神经网络，代码如下

```
def tanh(x):
    return (np.exp(x)-np.exp(-1*x))/(np.exp(x)+np.exp(-1*x))
def dtanh(y):
    return 1.0 - y*y

class NN:
    def __init__(self, ni, nh_li, no):
        # 输入层、隐藏层列表、输出层
```

```
9           self.ni = ni
10          self.nh_li = nh_li
11          self.nh_len = len(nh_li)
12          self.no = no
13          self.w = []
14
15          self.w.append(np.random.normal(0.0, pow(self.nh_li[0],-0.5),
16                                         (self.nh_li[0],self.ni)))
17          for i in range(self.nh_len-1):
18              self.w.append(np.random.normal(0.0,
19              pow(self.nh_li[i+1],-0.5), (self.nh_li[i+1],self.nh_li[i]))
20          self.w.append(np.random.normal(0.0, pow(self.no,-0.5),
21                                 (self.no,self.nh_li[self.nh_len-1])))
22
23      def oneRound(self, inputs,targets,N):
24          targets = np.array(targets,ndmin=2).T
25          # 激活输入层
26          inputs = np.array(inputs,ndmin=2).T/255
27          # 隐含层输出
28
29          outputs = []
30          outputs.append(tanh(np.dot(self.w[0],inputs)))
31          for i in range(self.nh_len-1):
32              outputs.append(tanh(np.dot(self.w[i+1],outputs[i])))
33          # 输出层输出
34          outputs.append(tanh(np.dot(self.w[self.nh_len],
35                                             outputs[self.nh_len-1])))
36
37
```

```
38          deltas = []
39          #计算输出层的误差
40          output_deltas = dtanh(outputs[self.nh_len])*
41                                      (targets - outputs[self.nh_len])
42          deltas.append(output_deltas)
43          for i in range(self.nh_len):
44              deltas.append(dtanh(outputs[self.nh_len-1-i])*
45                   np.dot(self.w[self.nh_len-i].T, deltas[i]))
46
47
48          for i in range(self.nh_len):
49              self.w[self.nh_len-i] += N*np.dot(deltas[i],
50                           np.transpose(outputs[self.nh_len-i-1]))
51          self.w[0] += N*np.dot(deltas[self.nh_len],
52                                            np.transpose(inputs))
53
54          return outputs[self.nh_len]
55
56      def test(self, inputs_list,Y):
57          ans_lis=[]
58          le = len(Y)
59          ans = 0
60          for i in range(le):
61              inputs = np.array(inputs_list[i],ndmin=2).T/255
62              outputs = []
63              outputs.append(tanh(np.dot(self.w[0],inputs)))
64              for j in range(self.nh_len-1):
65                  outputs.append(tanh(np.dot(self.w[j+1],outputs[j])))
66              # 输出层输出
```

```
67              outputs.append(tanh(np.dot(self.w[self.nh_len],
68                                         outputs[self.nh_len-1])))
69              if np.argmax(outputs[self.nh_len])==Y[i]:
70                  ans+=1
71              ans_lis.append(np.argmax(outputs[self.nh_len]))
72          print('准确率',ans/le)
73          return ans/le,ans_lis
74
75      def weights(self):
76          return self.w
77
78      def train(self, X, Y,iterations=1, N=0.01):
79          len_total = len(Y)
80          for i in range(iterations):
81              for p in range(len_total):
82                  inputs = X[p]
83                  target = Y[p]
84                  targets = trans(target)
85                  self.oneRound(inputs,targets,N)
86  def trans(li):
87      re_list =[0.0]*10
88      re_list[li]=1
89      return re_list
```

我们分别训练三层、四层神经网络

```
1  # 三层
2  num=60000
3  n = NN(784, [89], 10)
4  n.train(X_train[:num],Y_train[:num],iterations=10)
```

```
acc_nn_3 ,predict_nn_3 = n.test(X_test,Y_test)
# 四层
num=60000
n = NN(784, [89,50], 10)
n.train(X_train[:num],Y_train[:num],iterations=10)
acc_nn_4 ,predict_nn_4 = n.test(X_test,Y_test)
```

得到的准确率分别如下 0.9479、0.9699

3.5 K 近邻

在 sklearn 中，与近邻法这一大类相关的类库都在 sklearn.neighbors 包之中。KNN 分类树的类是 KNeighborsClassifier，KNN 回归树的类是 KNeighborsRegressor。我们调用 KNN 分类树，并编写如下代码，k 值分别取 3 和 4

```
from sklearn.neighbors import KNeighborsClassifier

kNN_classifier_3= KNeighborsClassifier(n_neighbors=3)
kNN_classifier_3.fit(X_train, Y_train)
predict_knn_3 = kNN_classifier_3.predict(X_test)
acc_knn_3=sum(predict_knn_3 == Y_test)/len(Y_test)

kNN_classifier_4= KNeighborsClassifier(n_neighbors=4)
kNN_classifier_4.fit(X_train, Y_train)
predict_knn_4 = kNN_classifier_4.predict(X_test)
acc_knn_4 =sum(predict_knn_4 == Y_test)/len(Y_test)
```

得到的准确分别如下 0.9705、0.9682

3.6 集成

对于分类问题的预测，我们通常使用的是投票法。最简单的投票法是相对多数投票法，也就是我们常说的少数服从多数，我们进行简单的投票操作，猜测最终集成的准确率应该大于个体学习器最高的 0.9705 的准确率。编写如下代码

```
ans = []
for i in range(10000):
    lab=[0]*10
    lab[int(predict_svm_liner[i])]+=1#0.9128
    lab[int(predict_svm_polynomial[i])]+=1#0.9577
    lab[predict_bayes_g[i]]+=1#0.5558
    lab[predict_bayes_b[i]]+=1#0.8423
    lab[predict_bayes_m[i]]+=1#0.8364
    lab[predict_knn_3[i]]+=1#0.9705
    lab[predict_knn_4[i]]+=1#0.9682
    lab[predict_nn_3[i]]+=1#  0.9479
    lab[predict_nn_4[i]]+=1#0.9696
    ans.append(lab.index(max(lab)))
sum(ans == Y_test)/len(Y_test)
```

得到的准确率为 0.9642。发现这和我们的预期有点差距，集成后的结果甚至比很多个体学习器的准确率还要低，于是怀疑比较差的分类器对我们的最终结果起的比较大的影响，因此将每个个体学习器的准确率作为他们做出的判断的权重，再次进行集成操作，代码如下

```
ans = []
for i in range(10000):
    lab=[0]*10
    lab[int(predict_svm_liner[i])]+=acc_svm_liner
    lab[int(predict_svm_polynomial[i])]+=acc_svm_polynomial
```

```
    lab[predict_bayes_g[i]]+=acc_bayes_g
    lab[predict_bayes_b[i]]+=acc_bayes_b
    lab[predict_bayes_m[i]]+=acc_bayes_m
    lab[predict_knn_3[i]]+=acc_knn_3
    lab[predict_knn_4[i]]+=acc_knn_4
    lab[predict_nn_3[i]]+=acc_nn_3
    lab[predict_nn_4[i]]+=acc_nn_4
    ans.append(lab.index(max(lab)))
sum(ans == Y_test)/len(Y_test)
```

这次得到的准确率为 0.9666，比直接投票提高了一点点，但还是不理想。

绝对多数投票法，也就是我们常说的要票过半数。在相对多数投票法的基础上，不光要求获得最高票，还要求票过半数。否则会拒绝预测，编写如下代码测试

```
ans = []
max_val=[]
for i in range(10000):
    lab=[0]*10
    lab[int(predict_svm_liner[i])]+=1#0.9128
    lab[int(predict_svm_polynomial[i])]+=1#0.9577
    lab[predict_bayes_g[i]]+=1#0.5558
    lab[predict_bayes_b[i]]+=1#0.8423
    lab[predict_bayes_m[i]]+=1#0.8364
    lab[predict_knn_3[i]]+=1#0.9705
    lab[predict_knn_4[i]]+=1#0.9682
    lab[predict_nn_3[i]]+=1#  0.9479
    lab[predict_nn_4[i]]+=1#0.9696
    ans.append(lab.index(max(lab)))
    max_val.append(max(lab))
```

```
sum(ans == Y_test)/len(Y_test)
anss = 0
ansst = 0
for i in range(10000):
    if max_val[i] >4.5:
        ansst+=1
        if ans[i] == Y_test[i]:
            anss+=1
print(anss/ansst)
```

因为训练了 9 个个体分类器，因此我们把阈值设为 5。最终的到 0.9791 的准确率，即当超过半数分类器判断相同时，这个预测正确率为 0.9791.

通过进一步学习了解到，当所有的分类器都独立运行的很好、不会发生有相关性的错误的情况下，集成后的准确率才会有较好的提升，然而每一个分类器都在同一个数据集上训练，导致其很可能会犯同一种错误，所以也会有很多票投给了错误类别导致集成的准确率下降。

于是尝试对每一个个体判别器采取随机的 10000 个训练样本，并最终进行进行集成测试，然而的到的结果也并不如预期的好，集成后的准确率略低于个体分类器的最高准确率。（具体代码见附件）

当我们把准确率较低的贝叶斯类分类器去掉后，再次测试

```
ans = []
for i in range(10000):
    lab=[0]*10
    lab[int(predict_svm_liner[i])]+=acc_svm_liner
    lab[int(predict_svm_polynomial[i])]+=acc_svm_polynomial
    lab[predict_knn_3[i]]+=acc_knn_3
    lab[predict_knn_4[i]]+=acc_knn_4
    lab[predict_nn_3[i]]+=acc_nn_3
    lab[predict_nn_4[i]]+=acc_nn_4
```

```
10      ans.append(lab.index(max(lab)))
11  sum(ans == Y_test)/len(Y_test)
```

可以得到最后的准确率达到 0.9739，比个体判别器最好的准确率高一点。可以看到不好的分类器对我们最后的影响还是很大的。

3.7 结果分析

通过上边的实验，我们发现最终集成的结果并不是很满意。在对数据进行训练时，这些分类器并不是独立的，它们会犯相同的错误，因为许多分类器是线性模型，它们最终的投票不会改进模型的预测结果。当尝试将数据分成几部分，每个部分的数据训练一个模型，也许会因为数据量不足导致训练出来的模型泛化能力较差。或许 Bagging 和 Boosting 方法是比较好的选择。

4 总结与收获

集成学习本身不是一个单独的机器学习算法，而是通过构建并结合多个机器学习器来完成学习任务集成学习可以用于分类问题集成，回归问题集成，特征选取集成，异常点检测集成等等，可以说所有的机器学习领域都可以看到集成学习的身影。此次实验中自己也发现了很多问题，更有助于对概念的理解。