数据分析及实践Exp4 Report

1. 实验描述

1.1 实验背景：

英雄联盟（League of Legends，LOL）是美国游戏开发商Riot Games（2011年由腾讯收购）开发和发行的一款多人在线战斗竞技游戏。

在游戏中，玩家扮演一个"召唤师"角色，每个召唤师控制一个拥有独特技能的"英雄"，并与一组其他玩家或电脑控制的英雄战斗。游戏的目标是摧毁对方的防御塔和基地。

召唤者峡谷是英雄联盟中最受欢迎的地图，在这种地图类型中，两队五名玩家竞争摧毁一个被称为基地的敌人建筑，这个建筑由敌人的队伍和一些防御塔护卫。每个队伍都希望保卫自己的建筑，同时摧毁对方的建筑。这些主要包括:

- Towers（防御塔）：每支队伍总共有11座防御塔
- Inhibitor（水晶）：每条道有一个水晶
- Dragon（大龙）
- Rift Herald（峡谷先锋）
- Baron Nasho（纳什男爵）
- Nexus（基地）

英雄联盟最具争议的元素之一，就是其严重的滚雪球效应。许多职业选手接受赛后采访时都提到其输赢都因为“滚雪球”，我们研究各方面各指标的数据，来看这些因素的发展是否真的影响了比赛的成败。在这个实验中，我们分析了8万场英雄联盟的排名比赛，尝试得出有效的结论。

1.2 实验要求：

Part1:

- 基于Exp3，手动实现一种分类算法（例如，决策树、 KNN或者朴素贝叶斯。并参考Exp3的特征工程，测试算法在 LOL数据集上的预测性能，撰写实验报告。

- 代码实现只允许使用 `numpy`、 `pandas` 库和 python内置库，不允许使用现有的机器学习库。
- 预测任务与实验三一致，以准确率作为评价指标。自行在 LOL数据集上划分训练集和验证集（4:1比例、交叉验证），汇报算法在验证集上的性能。

Part2：

- 基于Exp3，预测测试集中每个样本的比赛持续时间 `gameDuration` 。
- 将数据集中一部分样本的标签，作为训练集，而另一部分样本作为测试集。
- 可以利用开源工具包， 也可以参考Exp3的数据分析与特征工程。
- 以均方误差 (Mean-square Error)作为评价指标。

1.3 数据集说明：

- `team1_win` 代表team1是否取得胜利
- `team1_firstBlood` 代表team1是否取得一血，其他类似特征同理
- `player1_championId` 代表玩家1选择的英雄ID
- `player1_kills` 代表玩家1的击杀数，其他类似特征同理
- `player1` ~ `player5` 属于 `team1` ， `player6` ~ `player10` 属于 `team2`
- `gameDuration` 代表游戏时长

2. Part1

2.1 算法选择

使用**K近邻算法**(K-Nearest Neighbor，KNN)，基本思想是一个样本与数据集中的k个样本的“距离”最近， 如果这k个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。KNN算法的关键在于k值的选取和“距离”计算的方式。

此处选择的“距离”为特征之差，选取的特征有：一血、一龙、一塔、一男爵、一先锋、一水晶、同时一血一龙、同时一龙一先锋、同时一男爵一先锋、同时一塔一水晶、场均击杀差、场均死亡差、场均助攻差、经济差、KDA差共15个特征。

最初选用的距离计算方式为：各特征差的绝对值，乘以基于Exp3得出的经验权重（或与 `win` 的相关系数），最后再对各加权特征差求和，作为该样本间的距离。后来发现由于数据量过大，该计算方式会极大增加计算时间，所以选择以放弃少量精度为代价，来提高计算速度。

改进后的距离计算方式为：最初将训练集每一个样本的特征加权求和作为该样本的坐标，将多维问题降为一维问题。计算距离只需要对加权特征和作差即可。理论上该举措会降低精度，实际实验结果表明精度的变化在0.2%左右，以此为代价来极大地提高计算速度是值得的。

2.2 优化设计

KNN算法虽然是最简单的分类算法之一，但其计算量往往是巨大的。在最初版本中，在1/5的测试集上预测的时间高达900秒，这是不可接受的。于是对算法的计算进行优化。

- 首先在距离计算方式上进行改进，将多维计算变为一维计算，只需要计算样本间的加权特征和之差。
- 预处理时，生成含有每个样本特征的 `dataframe` ，其中包含已归一化的特征值，所有特征值位于0~1之间，其中将一血类的 `bool` 值转化为 `int` 值，并放缩100倍（即0-0.01）。同时将所有特征的加权和（即坐标）也存入其中。
- 完成测试任务时，将测试集的 `dataframe` 转化为 `array` 来操作。
- 运用 `array` 的 `lexsort()` 函数来进行距离升序排序，并选取前k个。
- 运用 `dataframe` 的 `value_counts()` 函数进行统计，选取出现次数最多的标签。

2.4 核心代码

```
#预处理后的数据数为75201，进行分组
h = int(75201/5)
H = [0,h,2*h,3*h,4*h,75201]
#随机打乱
df=df.reindex(np.random.permutation(df.index))
accuracy = []
for i in range(5):
    #划分测试集与训练集
    test = df[H[i]:H[i+1]]
    train = pd.concat([df[0:H[i]], df[H[i+1]:75201]], axis=0)
    #生成特征表
    train = gettrain(train)
    train_f = getfeature(train,train)
    test_f = getfeature(train,test)
    #测试数据个数、k值选取
    right = 0
    n = test.shape[0]
    k = 37
```

```
    #转化为array
    t0 = np.array(train_f[['sum','win']])
    t1 = np.array(test_f['sum']).tolist()
    t2 = np.array(test_f['win']).tolist()
    #开始测试
    time_start=time.time()
    for j in range(n):
        t3 = abs(t0 - [t1[j],0])    #距离
        idex=np.lexsort([t3[:,0]])
        temp = pd.DataFrame(t0[idex, :][0:k])[1]    #排序取前k个
        result = temp.value_counts().index[0]   #统计取第一个
        if result == t2[j]: #准确率统计
            right += 1
    accuracy  += [right/n]
    time_end=time.time()
    print('Fold',i+1,':','准确率为: %.3f'%(accuracy[i]),'耗时: %.6f s'%(time_end-time_start))

sum = 0
for i in accuracy:
    sum = sum + i
avg_acc = sum/5
print('平均准确率: %.3f'%(avg_acc))
```

2.3 结果分析

```
Fold 1 : 准确率为: 0.977 耗时: 81.188232 s
Fold 2 : 准确率为: 0.977 耗时: 81.058267 s
Fold 3 : 准确率为: 0.975 耗时: 80.908459 s
Fold 4 : 准确率为: 0.978 耗时: 81.801882 s
Fold 5 : 准确率为: 0.979 耗时: 81.275309 s
平均准确率: 0.977
```

平均准确率为97.7%，是一个可观的结果，但每一次验证的平均时常约为81.2秒，尽管从900多秒到81.2秒已经经过了很多优化，但对于后续调参来说还是很不方便。

2.4 算法改进：类重心KNN

在传统的KNN算法中，为了找到这个含有k个训练集中的样本的最近邻，需要计算该未知样本点和所有训练集样本之间的距离，然后从最小距离开始计样本数，一直计算到有K个样本数为止。

有一种简化的算法称为类重心法[1]，即将训练集中每类样本的重心求出，然后判别位置样本点与各类重心的距离，未知样本点距哪一类重心距离最近，位置样本就属于哪一类。

本Part共两类——team1胜、team2胜。于是在求的特征表的基础上，计算两类样本的重心，即类特征均值，计算测试样本与两个重心的距离，极大的提升了计算时间，并且对精度的影响并不大。

```
accuracy = []
for i in range(5):
    #划分测试集与训练集
    test = df[H[i]:H[i+1]]
    train = pd.concat([df[0:H[i]], df[H[i+1]:75201]], axis=0)
    #生成特征表
    train = gettrain(train)
    train_f = getfeature(train,train)
    test_f = getfeature(train,test)
    #重心计算
    train_f_1 = train_f[train_f['win']==1]
    train_f_0 = train_f[train_f['win']==0]
    center_1 = train_f_1.sort_values(by = ['sum'])
[(train_f_1.shape[0]//2):(train_f_1.shape[0]//2+1)]
['sum'].values[0]
    center_0 = train_f_0.sort_values(by = ['sum'])
[(train_f_0.shape[0]//2):(train_f_0.shape[0]//2+1)]
['sum'].values[0]

    right = 0
    n = test.shape[0]
    #转化为array
    t1 = np.array(test_f['sum']).tolist()
    t2 = np.array(test_f['win']).tolist()
    #开始测试
    time_start=time.time()
    for j in range(n):
        if abs(center_1-t1[j]) <= abs(center_0-t1[j]): #与重心
距离
```

```
            result = 1
        else:
            result = 0
        if result == t2[j]:
            right += 1
    accuracy  += [right/n]
    time_end=time.time()
    print('Fold',i+1,':','准确率为: %.3f'%(accuracy[i]),'耗时: %.6f s'%(time_end-time_start))

sum = 0
for i in accuracy:
    sum = sum + i
avg_acc = sum/5
print('平均准确率: %.3f'%(avg_acc))
```

结果：

```
Fold 1 : 准确率为: 0.978 耗时: 0.017894 s
Fold 2 : 准确率为: 0.978 耗时: 0.019946 s
Fold 3 : 准确率为: 0.978 耗时: 0.019946 s
Fold 4 : 准确率为: 0.978 耗时: 0.028446 s
Fold 5 : 准确率为: 0.977 耗时: 0.020291 s
平均准确率: 0.978
```

平均准确率为97.8%，并且计算时间减少至10^{-2}级，是比较可观的。

3. Part2

2.1 算法选择

分别选用BP单层神经网络、KNN、SVM、决策树、Bayes、多元回归、MLP、随机森林算法进行测试。

2.2 预处理

- 选取两队杀敌总数、死亡总数、助攻总数、经济和共四个特征，做归一化。
- 将两个csv文件合并，取后60000个用于训练与测试，前20000个来预测。
- 将时间标签化（除了多元回归）

```
def roundup(x):
    return int(math.ceil(float(x) / 35)) *35
```

2.3 算法与结果

2.3.1 BP单层神经网络

```
##BP单层神经网络
import tensorflow.compat.v1 as tf
#定义参数
d=4 #输入节点个数
l=1 #输出节点个数
q=2*d+1 #隐层结点个数,采用经验公式2d+1
eta=0.5 #学习率
error=0.0016    #精度

#初始化权值和阈值
w1= tf.Variable(tf.random.normal([d, q], stddev=1, seed=1))
#seed设定随机种子，保证每次初始化相同数据
b1=tf.Variable(tf.constant(0.0,shape=[q]))
w2= tf.Variable(tf.random.normal([q, l], stddev=1, seed=1))
b2=tf.Variable(tf.constant(0.0,shape=[l]))

#输入占位
tf.compat.v1.disable_eager_execution()
x = tf.placeholder(tf.float32, shape=(None, d))
y_ = tf.placeholder(tf.float32, shape=(None, l))

#前向传播
a=tf.nn.sigmoid(tf.matmul(x,w1)+b1) #sigmoid激活函数
y=tf.nn.sigmoid(tf.matmul(a,w2)+b2)
mse = tf.reduce_mean(tf.square(y_ -  y))    #损失函数采用均方误差
train_step = tf.train.AdamOptimizer(eta).minimize(mse)  #Adam算法

data = totalX
labels = totalY
labels = (labels - Min[4])/(Max[4]- Min[4])

#创建会话来执行图
```

```
with tf.Session() as sess:
    init_op = tf.global_variables_initializer()#初始化节点
    sess.run(init_op)

    STEPS=0
    while True:
        sess.run(train_step, feed_dict={x: data, y_: labels})
        STEPS+=1
        train_mse= sess.run(mse, feed_dict={x: data, y_: labels})
        if STEPS % 10 == 0:#每训练100次，输出损失函数
            print("第 %d 次训练后,训练集损失函数为：%g" % (STEPS, train_mse))
        if train_mse < error:
            break
    print("总训练次数：",STEPS)

    #测试
    Normal_y= sess.run(y, feed_dict={x: testX})#求得测试集下的y计算值
    DeNormal_y=Normal_y*(Max[4] - Min[4])+Min[4] #将y反归一化
    Dy = DeNormal_y.tolist()
    DY = testY.tolist()
    n = len(Dy)
    right = 0

    for i in range(n):
        if Dy[i][0] >= (math.floor(DY[i][0]/100)*100-150) and Dy[i][0] <= (math.ceil(DY[i][0]/100)*100+150):
            right += 1
    print('正确率为：',right/n)
    err_BP = sess.run(mse, feed_dict={y: DeNormal_y, y_: testY})#计算均方误差
    print("测试集均方误差为：",err_BP)

    #预测
    test = np.array(Test)
    y= sess.run(y, feed_dict={x: test})
    y = y*(Max[4] - Min[4])+Min[4] #将y反归一化
```

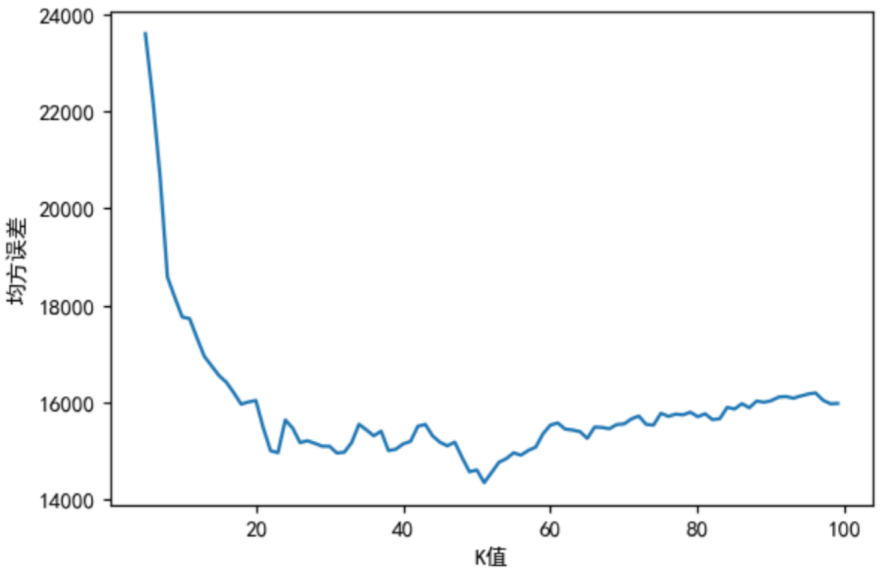
结果：

```
总训练次数：  478
正确率为：  0.95175
测试集均方误差为：  16664.203
```

2.3.2 KNN

```python
##KNN
from sklearn import neighbors
data = totalX
labels = totalY
E = [1000000,0]
err_KNN = [0] * 95

#遍历寻找最佳k值
for k in range(5,100):
    knn = neighbors.KNeighborsClassifier(k)
    knn.fit(data, labels)
    y = knn.predict(testX)
    #均方误差
    err_KNN[k-5] = metrics.mean_squared_error (y, testY)
    if err_KNN[k-5] < E[0]:
        E[0] = err_KNN[k-5]
        E[1] = k
        Y = y

n = len(y)
right = 0
for i in range(n):
    if Y[i] >= (math.floor(testY[i][0]/100)*100-150) and Y[i]
<= (math.ceil(testY[i][0]/100)*100+150):
    #if abs(Y[i]-testY[i][0]) <= 100:
        right += 1

plt.figure(dpi=110)
plt.xlabel("K值")
plt.ylabel("均方误差")
plt.plot(range(5,100),err_KNN)
plt.show()
print('最佳正确率为：',right/n)
print("最佳K值为：",E[1],"测试集均方误差为：",E[0]);
```

结果：



最佳正确率为： 0.9745833333333334
最佳K值为： 51 测试集均方误差为： 14351.010333333334

2.3.3 SVM

```
#SVM
from sklearn import svm
data = totalX
labels = totalY
clf = svm.SVR()
clf.fit(totalX, totalY)
y = clf.predict(testX)
err_svm = metrics.mean_squared_error (y, testY)
print("测试集均方误差为：",err_svm);
```

结果：

测试集均方误差为： 23766.188498855518

2.3.4 决策树

```
##决策树
from sklearn import tree
data = totalX
labels = totalY
clf = tree.DecisionTreeRegressor()
clf = clf.fit(data, labels)
y = clf.predict(testX)
err_tree = metrics.mean_squared_error (y, testY)
print("测试集均方误差为：",err_tree);
```

结果：

测试集均方误差为： 24150.702625

2.3.5 贝叶斯

```
#贝叶斯
from sklearn.naive_bayes import GaussianNB
data = totalX
labels = totalY
gnb = GaussianNB()
gnb = gnb.fit(data, labels)
y = gnb.predict(testX)
err_bayes = metrics.mean_squared_error (y, testY)
print("测试集均方误差为：",err_bayes)
```

结果：

测试集均方误差为： 52772.974916666666

2.3.6 多元回归

```
#要重新预处理，时间不预处理（不使用roundup函数）
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

data = totalX
labels = totalY

reg = PolynomialFeatures(degree=4)
data_train = reg.fit_transform(data)
```

```
testX_train = reg.fit_transform(testX)
lg = LinearRegression()
lg.fit(data_train,labels)
y = lg.predict(testX_train)
err_ploy = metrics.mean_squared_error (y, testY)
print("测试集均方误差为：",err_ploy)
```

结果：

测试集均方误差为： 12239.60481076169

2.3.7 MLP

```
from sklearn.neural_network import MLPClassifier

data = totalX
labels = totalY

clf = MLPClassifier(solver='lbfgs', alpha=1e-
5,hidden_layer_sizes=(3, 2), random_state=1)
clf.fit(data, labels)
y = clf.predict(testX)
err_mlp = metrics.mean_squared_error (y, testY)
print("测试集均方误差为：",err_mlp)
```

结果：

测试集均方误差为： 25560.91075

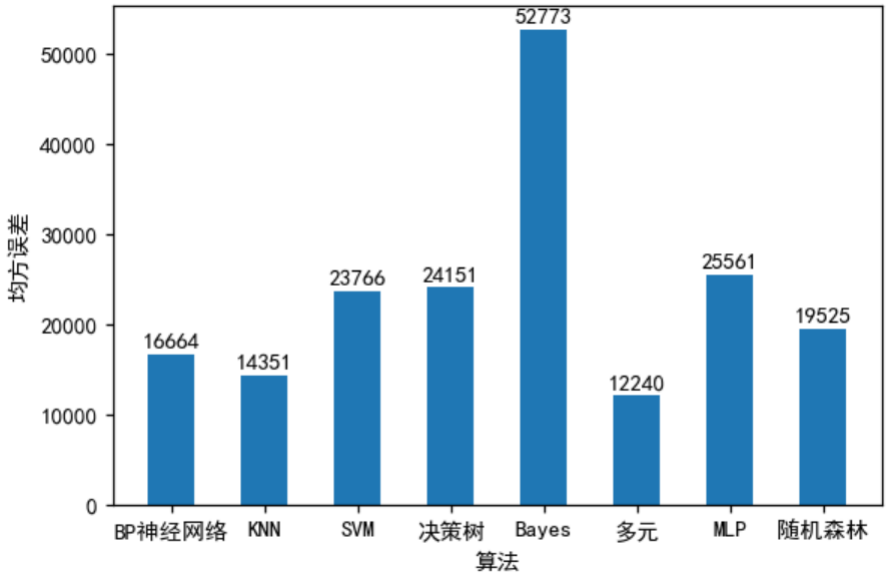
2.3.8 随机森林

```
from sklearn.ensemble import RandomForestClassifier

data = totalX
labels = totalY

clf = RandomForestClassifier(n_jobs=3)
clf.fit(data, labels)
y = clf.predict(testX)
err_for = metrics.mean_squared_error (y, testY)
print("测试集均方误差为：",err_for)
```

结果：

测试集均方误差为： 19525.337083333332

2.4 结果对比



在系统训练集、测试集下，均方误差最小的为**KNN算法**与**多元回归算法**。

2.5 Autogluon库

```
from autogluon.tabular import TabularDataset,TabularPredictor
predictor =
TabularPredictor(label='gameDuration').fit(df[0:48000])
a=df[48000:60000].drop(['gameDuration'],axis=1)
y = predictor.predict(a)
err_auto = metrics.mean_squared_error (y, testY)
print("测试集均方误差为: ",err_auto);
```

结果：

测试集均方误差为： 11266.0005795075

准确度最佳。

参考文献

[1] 宋毅飞,周剑秋.KNN算法与其改进算法的性能比较[J].机电产品开发与创新,2017,30(02):60-63.