2018-2-9

机器学习算法解决银行市场问题

决策树、逻辑回归、SVM 算法、Kmeans 算法

李天幼
浙江大学 计算机科学与技术学院

目录

一、 Bank Marketing

1、 问题描述

问题来源于葡萄牙的一项银行运动。银行通过联系大量的用户，调查用户的基本情况，来确定这个用户是否会在这个银行内存款。银行询问用户的问题即是要考虑的特征。当然某些特征对于 classification 问题不合适，考虑到机器的性能，这些特征应该去掉。

2、 特征选取

数据来源于 UCI: Bank Marketing Data Set（https://archive.ics.uci.edu/ml/datasets/bank+marketing）。实例共有 45211 个，特征共有 20 个，属于 classification 问题。特征包括：

1、Age：年龄。

2、job：工作。
'admin.','bluecollar','entrepreneur','housemaid','management','retired','self-employed','services','student','technician','unemployed','unknown'

3、Marital：婚姻情况。'divorced','married','single','unknown'

4、education ：受教育水平。
'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown'

5、default: 有没有信用卡。 'no','yes','unknown'

6、housing: 是否有房子 。'no','yes','unknown'

7、loan: 是否有个人贷款 。'no','yes','unknown'

※8、contact: 联系工具。'cellular','telephone'

※9、mon th: 最近联系的月份。 'jan', 'feb', 'mar', ..., 'nov', 'dec'

※10、day_of_week: 最近联系的星期。'mon','tue','wed','thu','fri'

11、duration：上次通讯时长。这个特征严重影响结果！

※12、campaign： 通讯次数。

※13、pdays：上次通讯之后的天数。999 代表还没联系过。

※14、previous：在这次银行运动之前，联系的次数。

15、poutcome：这次银行运动的结果。'failure','nonexistent','success'

16、emp.var.rate：工作变更概率。

17、cons.price.idx：消费水平。

18、cons.conf.idx： 消费者信心指数。月度指标，数字。Consumer Confidence Index, CCI，由问卷调查法得出。反应消费者信心强弱，并进行量化，评价包括当前经济形势，收入水平，消费心理等等。

19、euribor3m：每三个月的欧元同业拆借利率。日常指标，数字。

※20、nr.employed：员工个数。

年龄、工作、婚姻状况、受教育水平是一个人的基本属性，也决定了一个用户有多少资金，是否将在银行存钱。受教育水平、有没有信用卡、有没有房子、有没有个人房贷属于一个人的高级属性，往往和一个人的工作、社会地位有关系。生活水平比较高的人更容易在银行存钱。根据 UCI 数据集，通讯时长是一个很关键的特征，将严重影响结果。其他的特征是一些指数，例如工作变更概率、消费水平、消费信心指数、欧元银行同业拆借利率等等。

根据数据集的描述，银行与用户的通话时长会严重影响结果，这在决策树算法中得到了体现。我们小组认为，银行与用户沟通的时间会影响用户的心理，时间越长，表明这个用户乐意和银行沟通，越信赖这家银行所提供的服务，自然在银行存款的概率越高。

同业拆借利率指金融机构同业之间的短期资金借贷利率。它有两个利率，拆进利率表示金融机构愿意借款的利率；拆出利率表示愿意贷款的利率。

同业拆借利率是拆借市场的资金价格，是货币市场的核心利率，也是整个金融市场上具有代表性的利率，它能够及时、灵敏、准确地反映货币市场乃至整个金融市场短期资金供求关系。当同业拆借率持续上升时，反映资金需求大于供给，预示市场流动性可能下降，当同业拆借利率下降时，情况相反。

同业拆借市场按有无中介机构参与可分为两种情况，即直接交易和间接交易，并由此导致不同的拆借利率的形成方式。在直接交易情况下，拆借利率由交易双方通过直接协商确定；在间接交易情况下，拆借利率根据借贷资金的供求关系通过中介机构公开竞价或从中撮合而确定，当拆借利率确定后，拆借交易双方就只能是这一既定利率水平的接受者。

二、 决策树

1、 算法

决策树算法的思想其实在日常代码中是常用的，任何的 if-else 语句都是决策树算法思想的一种体现。决策树更在意把哪个条件放在根节点，按照影响结果的重要程度进行排序。为了解决把哪个条件放在根节点的问题，引入了熵的概念。

关于熵，有以下几个概念：信息量，指判断出一个事物的正确性，需要询问是和否的次数；信息熵，代表混乱程度，信息熵越大，信息的混乱程度越大，由 Quinlan 提出，以 Shannon 的信息论为依据；信息论，熵的计算公式

$$I = -\sum_{i=1}^{n} p(x_i, y_i) log p(x_i, y_i)$$

熵增益，指事件 x_i 发生之前和发生之后，熵的该变量。信息增益=I（前）-I（后）。信息增益高，说明分类器分的好，能大大降低数据的信息量。条件熵，是在已知某个事件发生的概率下，另一个事件发生后的熵

$$H(X|Y) = -\sum_{i=1}^{n} p(x_i, y_i) log p(x_i|y_i)$$

Log 是以 2 为底的对数，但是这样运算会使熵大于 1。在我们小组的学习过程中，我们想使熵在（0,1）之间，所以底数选择使用类别的个数。这样做虽然没有理论的依据，但是在计算熵增益的时候，信息熵都是统一的，所以不会影响结果。我们和其他组交流的时候，其他组也这样认为。

先算出信息熵，概率用训练集的比值来代替。然后计算每次增加一个特征的熵增益，熵增益最大的特征，就是最能区分数据的特征，作为决策树的根节点。在计算第二个节点的时候，要先认为已经对根节点做出了选择，所以信息熵要重新计算，在已知根节点选择情况的条件下计算。

2、 代码实现

```
def main(sc):
    iris_lines = sc.textFile("bank5.csv")
    #print(iris_lines.collect())

    iris_lines = iris_lines.map(lambda item: item.split(","))
    #print(iris_lines.collect())

    iris_points = iris_lines.map(lambda item:tokenize(item))
    #print(iris_points.collect())
    print(iris_points)

    splits = iris_points.randomSplit([0.7, 0.3],11)
    training = splits[0]
    testing = splits[1]
    print(training.count())
    print(testing.count())

    #model = DecisionTree.trainClassifier(iris_points, numClasses =
3,maxDepth = 5,maxBins = 32,{})
    model = DecisionTree.trainClassifier(training, 3,{},maxDepth=13)
    print(model)
```

```
def tokenize(item):
    vector =
Vectors.dense(float(item[0]),float(item[1]),float(item[2]),float(item[3]),
float(item[4]),float(item[5]),float(item[6]),float(item[10]),float(item[14
]),float(item[15]),float(item[15]),float(item[16]),float(item[17]),float(i
tem[18]))
    if item[20] == "20": #label=no
        label = 0.0
    elif item[20] == "21": #label=yes
        label = 1.0
    else:
        label = 2.0

    item = LabeledPoint(label,vector)
    return item
```

使用了 python 的 pyspark 库。

代码中省略的 pyspark 的入口部分。要使用 pyspark，首先应该建立一个 SparkContext 的对象 sc，参数是 SparkConf 对象。Sc 在 main 函数中可以进行对数据集的各种操作。

textFile()函数读取文件“bank5.csv”。读进来的文件是一个一维数组，每个元素都是一个 RDD。每个元素都是一行数据，包括所有 Feature 和一个 label，用逗号分隔。

Item.split()函数用逗号分隔每个数据，形成一个新的二维数组。

定义一个 tokenize 函数，识别 iris_lines 数组中的各个 features 和 label，形成 labeledpoint 数组。在 tokenize 函数中，我们手动定义了选择哪些特征，不需要用到的特征就没有放入 labeledpoint 里。Label 有两种，“20”代表“no”，“21”代表“yes”，代码里写了第三种 class，这个其实可有可无。

下面 iris_points.randomSplit()函数用于将数据集划分成训练集和测试集。我们采用 0.7,0.3 的划分比例。经过尝试，这个比例比较合适，再小的话训练出的树不准确，再大的话就过拟合。形成了两个 labeledpoint，training 和 testing。

DecisionTree.trainClassifier()函数是决策树构建。把训练集 training 作为参数，默认 class 的数量是 3，由于我们使用了 13 个特征，所以决策树的最大深度设置为 13，如果这个数字小于特征数量的话，代码运行中会产生异常。产生的决策树对象是 model。

Model.predict()函数用来做预测。将样本的 label 与决策树的 label 作比较。结果放入 predictionList 中。

用 MulticlassMetrics()方法得到混淆矩阵 metrics。然后用 metrics.precision()方法得到精确度，metrics.accuracy()方法得到准确率，metrics.recall()方法得到召回率。

至此，用 pyspark 库完成了决策树的预测。Pyspark 提供了完整的方法，使用起来非常方便。决策树存在过拟合的问题，当决策树的节点过多时，会发生过拟合。过拟合是指训练的条件太过严格，训练集的数据严格符合每个特征，但是换一个相似的数据就识别错误了。决策树在处理连续字段时存在困难，如何把连续字段变成离散变量，即怎样选取分割点是一个很重要的问题。决策树在处理相关性比较强的特征时效果不好，最好将变量的相关性做计算，排除掉相关性比较大的特征。

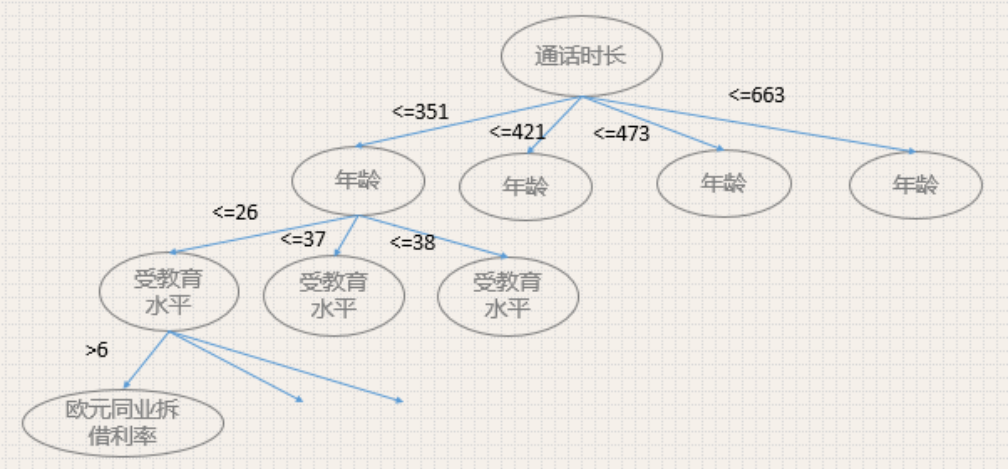
3、 结果

准确率：

```
精确度:0.9566835871404399
准确率:0.9566835871404399
/usr/local/spark-2.2.0-bin-h
accuracy.
  warnings.warn("Deprecated
召回率:0.9566835871404399
```

可以看到，准确率=95.67%，这个准确率还是非常高的。经过我们不断的测试，这是我们的代码能达到的最高的准确率。UCI 数据集中提到，通话时长是最重要的因素，当我们去掉通话时长这个特征的时候，准确率降低到 93%左右，这也验证了 UCI 数据集的说法。

生成的决策树



这个决策树以通话时长作为根节点，再次验证了 UCI 数据集的说法。

由于决策树比较大，这里只画了一部分。

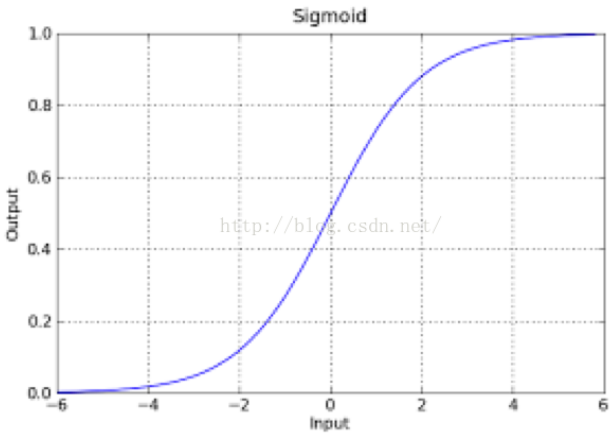
三、 逻辑回归

1、 算法

逻辑回归与线性回归类似。线性回归把一系列离散的点拟合成一条曲线，这条曲线可以做预测，给定自变量，能得到一个预测的因变量。逻辑回归想要得到一个[0,1]之间的数值，而不是(-∞,+∞)的数字，所以使用了一个 sigmoid 函数

$$g(z) = \frac{1}{1+e^{-z}}$$

图像：



代价函数 cost function：代价函数用于判定我们所取的 θ 是否合适，在给定 y 的条件下，我们取的 θ 与准确值相差的越远，那么代价就越大。逻辑回归的代价函数形如

$$Cost(h_\theta(x), y)\begin{cases} -\log\left(h_\theta(x)\right) & if\ y = 1 \\ -\log\left(1 - h_\theta(x)\right) & if\ y = 0 \end{cases}$$

也就是说，当类别 y=1 时，判定的 h θ (x)=1，则代价为 0，而 h θ (x)→0 时，代价趋近于正无穷。相反，当类别 y=-1 时也是这样。这个函数很好的描述了逻辑回归中的代价。

梯度下降：我们希望取到 θ 值，使代价最小。梯度下降算法是完成这个工作的一个很好的方法。为了找到函数的最小值点，我们应该朝着下降最快的方向前进，也就是导数绝对值最大的方向。在函数上找一个初值，每次朝着梯度最大的方向前进一个步长 α，重新审视这一点的导数，再次前进步长 α。

逻辑回归表示了某个样本属于某个类别的概率。在输出结果的大范围内，逻辑回归可以通过 sigmoid 函数映射到(0,1)，完成概率的估测。这是逻辑回归解决二分类问题的方法。传统的求解逻辑回归参数的方法是梯度下降，构造代价函数，然后用梯度下降算法向前迈进一小步，直至 N 次后到达最低点。

Softmax 回归：softmax 回归是逻辑回归模型在多分类问题上的推广。Softmax 代价函数和 logistic 代价函数在形式上非常类似，在 softmax 代价函数中对类的 k 个可能值进行了累加。我们使用迭代的优化算法求解最小化问题，比如梯度下降法或者 LBFGS 算法。我们通过便加一个权重衰减项来修改代价函数，这个衰减项会惩罚过大的参数值。

L-BFGS 算法：在 BFGS 算法的基础上，LBFGS 算法做了如下改进：在每一步估算 hesse 矩阵的近似的时候，给出一个当前的初始估计 H0；利用过去的 m-1 次的曲率信息修正 H0 直到得到最终的 hesse 矩阵。LBFGS 算法的优点是，它不需要记忆 H 或者 B 这两个近似矩阵，只需要存储{si,yi}的一个序列，这样就大大节省了存储空间。

2、 代码实现

```
def tokenize(item):
    vector =
Vectors.dense(float(item[0]),float(item[1]),float(item[2]),float(item[3])
,float(item[4]),float(item[5]),float(item[8]),float(item[9]),float(item[1
0]),float(item[14]),float(item[15]),float(item[16]),float(item[17]),float
(item[18]),float(item[19]))
    if item[20] == "20":
```

```
        label = 0.0
    elif item[20] == "21":
        label = 1.0
    else:
        label = 2.0

    item = LabeledPoint(label,vector)
    return item

## Main functionality

def main(sc):
    iris_lines = sc.textFile("bank5.csv")
    #print(iris_lines.collect())

    iris_lines = iris_lines.map(lambda item: item.split(","))
    #print(iris_lines.collect())

    iris_points = iris_lines.map(lambda item:tokenize(item))
    print(iris_points.collect())
    print(iris_points)

    splits = iris_points.randomSplit([0.6, 0.4], seed = 11)
    training = splits[0]
    testing = splits[1]

    model = LogisticRegressionWithLBFGS.train(training,iterations=120,
numClasses=3)

    predictionAndLabels = testing.map(lambda item:
[float(model.predict(item.features)), item.label])
    print(predictionAndLabels.collect())
  metrics = MulticlassMetrics(predictionAndLabels)
    print("精确度:"+str(metrics.precision()))
    print("准确率:"+str(metrics.accuracy))
    print("召回率/:"+str(metrics.recall()))
    print("混淆矩阵")
    print(metrics.confusionMatrix())

    sc.stop()
if __name__ == "__main__":
    # Configure Spark
```

```
    conf = SparkConf().setAppName(APP_NAME)
    conf = conf.setMaster("local[*]")
    sc   = SparkContext(conf=conf)
    # Execute Main functionality
    main(sc)
```

Sc.textFile()函数读取文件，形成一个 list，每个元素都是一个 RDD。

Item.split()函数用逗号分隔元素，形成一个二维数组

Item.tokenize()函数将 RDD 中的每一项变成一个 labeledoint，一个 label 后面跟着一个 list，包括所有的特征。

Iris_points.randomSplit()函数随机将数据分为训练集和测试集，我们采用(0.6,0.4)的划分。随机种子=11，用于产生真随机数。

LogisticRegressionWithLBFGS.train()函数是逻辑回归构建的核心函数。这个函数采用了 softmax 回归模型，用来处理逻辑回归的多分类问题。代价函数采用梯度下降的算法求最小值。参数 training 是训练集，iterations 是迭代次数，用于梯度下降时前进的次数。这里的 numClass 是 3，其实第三个类不是必要的。用这个函数得到 medel，即最终的逻辑回归模型。

Model.predict()函数用来做预测。将测试集的 features 传进去，生成的逻辑回归模型就会自动做预测。生成 predictionAndLabels 预测结果。

MulticalssMetrics()函数以 predictionAndLabels 做参数，生成矩阵。精确度=metrics.precision，准确率=metrics.accuracy，召回率=metrics.recall，混淆

矩阵=metrics.confusionMatrix。混淆矩阵的行是真实的类别，列的预测出的类别，所有对角线上的数字是预测正确的数目，而对角线之外的数字都是预测错的。通过混淆矩阵，我们就能知道把哪些数据预测错了，从而做出优化。

3、 结果

准确率

精确度:0.9665913087164029
准确率:0.9665913087164029

准确率是 96.66%，这个准确率也是相当可观。

四、 SVM 算法

1、 算法

Support vector machine 支持向量机算法。在平面中存在所有的数据的点，每个点都属于一个类别。找到一个超平面，能够分开所有的属于不同类别的点。例如

$$wx + b = 0$$

是一条直线，在平面中成功分隔了两种类型的点。

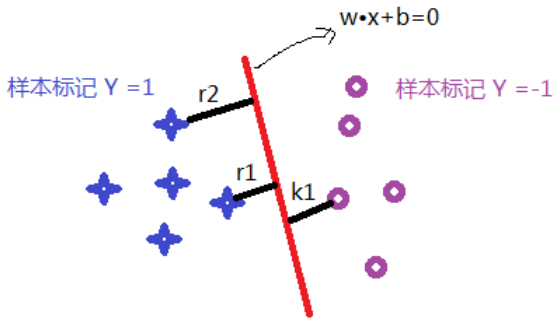


图 1-2

那么怎样计算 w 和 b 呢？最佳的 w 和 b 应该使离直线最近的点具有离直线最大的距离，这样的直线才算把点分成最远的两种类别。

支持向量到超平面的最小距离

$$\max[min\frac{y(wx+b)}{||w||}]$$

也就是计算最大化

$$max\frac{1}{||w||}$$

为了之后的求导和计算方便，这个式子等于计算最小化

$$min\frac{1}{2}||w||^2$$

我们认为超平面可以将样本正确分类，这样就变成了一个最优化问题

$$min\frac{1}{2}||w||^2\ subject\ to\ y_i(w^t x_i+b)\geq 1$$

拉格朗日函数：用拉格朗日函数来计算这个凸二次规划问题。拉格朗日函数更好解，也可以自然引入核函数，推广到非线性分类。构造拉格朗日函数

$$\psi(w,b,\alpha) \equiv \frac{1}{2}w^T w - \sum_{i=1}^{n} \alpha_i y_i (w \cdot x_i - b)$$

分别对 w 和 b 求导。代入拉格朗日函数后，得到原问题的对偶问题。**对偶问题**指的是每一个线性规划问题都伴随有另一个线性规划问题，称为这个问题的对偶问题。对偶问题能将最大化变为最小化，收益系数和右端常数能反过来，约束不等式符号相反，系数矩阵是转置关系，原始问题的约束方程数对应于对偶问题的变量数，等等特点。

核函数：核函数的思想是找到一个函数，这个函数使得在低维空间中进行计算的结果和映射到高维空间中计算内积的结果相同。这样避免了维度的爆炸性增长，又将数据集映射到了高维空间。常用的核函数有多项式核函数、高斯核核函数、线性核函数、字符串核函数。

SVM 算法中还存在噪声（noise）的问题，噪声指的是误差过大的数据点，这样的点的存在会影响本来正确的超平面结果。SVM 为了解决噪声的问题，允许数据点在一定程度上偏离超平面，约束就变为

$$y_i(w^T x_i + b) \geq 1 - \xi_i$$

其中$\xi \geq 0$，称为**松弛变量**（slack variable）。松弛变量不能太大，否则任意的超平面都变成符合条件的超平面。为此，引入**损失函数**

$$\sum_{i=1}^{N} \xi_i$$

还需要一个惩罚因子 C，代表对离群点带来的损失的重视程度，它的值越大，对目标函数的损失越大，意味着不愿意抛弃这些点。

结合这些概念，SVM 的优化问题变成

$$\min \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \ subject\ to\ y_i(wx_i + b) \geq 1 - \xi_i$$

再用拉格朗日函数得到对偶问题，这就是能处理线性和非线性情况，并且能容忍噪声的 SVM 算法。

支持向量机是非常受欢迎的二分类线性分类器。在处理多分类问题时，可以采用一对一的方式，即每次选择一个类作为正样本，另一个类作为负样本，如此重复，处理所有的样本。虽然需要 k*(k-1)/2 个分类器，但是不会出现数据偏斜的问题。

SGD 随机梯度下降算法：SGD 算法为了解决梯度下降法中，训练过程随着数据集的增大而变缓慢的问题。随机梯度下降是通过每个样本来迭代更新一次，如果样本量很大的情况，那么可能只用其中几万条或者几千条的样本。缺陷是这样做会导致准确度下降。

2、 代码实现

```
## Closure Functions
def tokenize(item):
    vector =
Vectors.dense(float(item[0]),float(item[1]),float(item[2]),float(it
em[3]),float(item[4]),float(item[5]),float(item[6]),float(item[7]),
float(item[8]),float(item[9]),float(item[10]),float(item[11]),float
(item[12]),float(item[13]),float(item[14]),float(item[15]),float(it
em[16]),float(item[17]),float(item[18]),float(item[19]))
    if item[20] == "20": #label=no
        label = 0.0
    elif item[20] == "21": #label=yes
        label = 1.0
    else:
        label = 2.0
```

```
    item = LabeledPoint(label,vector)
    return item

## Main functionality

def main(sc):
    bank_lines = sc.textFile("bank5.csv")
    bank_lines = bank_lines.map(lambda item: item.split(","))
    #print(bank_lines.collect())

    bank_points = bank_lines.map(lambda item:tokenize(item))
    print(bank_points)
 #print(iris_points.collect())
   # print(iris_points)

  splits = bank_points.randomSplit([0.5, 0.5],11)
    training = splits[0]
    testing = splits[1]
   # print(training.count())
   # print(testing.count())

    numIteration3 =100
    stepsize =1
    miniBatchFraction=1
    model =
    SVMWithSGD.train(training,numIteration3,stepsize,miniBatchFracti
    on)
    print(model)
    predictionAndLabel = training.map(lambda
    bank_points:(bank_points.label,model.predict(bank_points.feature
    s)))
    trainErr = predictionAndLabel.filter(lambda
predictionAndLabel:predictionAndLabel[0]!=predictionAndLabel[1]).cou
nt()/float(bank_points.count())
    print ("Accuracy = "+str(1-float(trainErr)))

    model.save(sc,"model")
    sameModel =SVMModel.load(sc,"model")
    err= sameModel.predict(bank_points.collect()[0].features)
    sc.stop()
```

```
if __name__ == "__main__":
    # Configure Spark
    conf = SparkConf().setAppName(APP_NAME)
    conf = conf.setMaster("local[*]")
    sc   = SparkContext(conf=conf)
    # Execute Main functionality
    main(sc)
```

Sc.textFile()函数读取 bank5.csv 文件。

Item.split()用逗号分隔读取的文件，形成一个二维数组 iris_lines。每个元素都是一个一维数组，包括所有的特征和一个 label。

用 tokenize()函数将 iris_lines 数组变成一个 labeledpoint 数组。Labeledpoint 数组中每个元素由一个 label 和一个特征列表组成。

Bank_points.randomSplit()函数将数据集划分为训练集和测试集，我们采用（0.5,0.5）的比例划分，经过不断尝试，这个划分比例是结果最好的。随机种子=11，用来产生随机数。训练集 training，测试集 testing。

设置迭代次数=100，步长=1，然后使用 SVMWithSGD.train()函数生成 SVM 模型。这是 SVM 模型生成的核心。SGD 算法上文简单介绍过，优点主要是减少迭代次数以加快速度。这个函数生成了 SVM 算法模型 model。

然后用 model.predict()函数进行测试。将测试结果放入 predictionAndLabel 中。

下面计算准确率。准确率可以用 1-错误率的方法获得，错误率是 trainErr 变量。trainErr 的计算方法是找出 predictionAndLabel 中，预测 label 与真实 label 不一致的数量，除以数据的总数。用 1-trainErr 即得到了准确率。

3、 结果

```
(weights=[-0.486984880927,-0.0643367255968,-0.0177140534764,-0.0385188661976,-0.
198659124421,-0.231320615732,-0.226932231941,-0.0117597806714,-0.0117892618023,-
0.0338674962283,19.2244147326,-0.032523892491,-11.7480208908,0.0,0.0,-0.01294460
30778,-1.10536271004,0.428215214547,-0.0572248424885,-61.0461152154], intercept=
0.0)
Accuracy = 0.9830056318545598
```

Weights 是算法生成的权值向量，表明了每个特征在算法中的权值，有些特征会被算法认为不重要，那么权值较低。其中，“通讯时长”的权值=19.22，是一个非常高的数字，再次验证 UCI 数据集的说法。

在我们多次的验证中，98.30%是我们能达到的最高准确率，一半训练集一半测试集。SVM 算法的参数使用默认。

在测试过程中，出现了一个我们不理解的现象，随着迭代次数的急剧变大，准确率反而严重降低。在梯度下降中，迭代次数的增加应该使目标点无限趋近于函数最小值点，准确率也应该趋近于一个稳定的最大值，这明显与代码运行结果不符。我们猜测，这是由于我们使用了 SGD 算法，而不是普通的梯度下降，SGD 算法会选择部分样本点下降，而不是全部，所以可能在这里 SGD 算法在很大的迭代次数中把误差也放大了。

五、 Kmeans 算法

1、 算法

Kmeans 聚类算法属于无监督学习，要点是按照数据内部的特征将数据集划分为不同的类别，类别内部数据很相似，类别之间的差别很大。重点在于计算样本之间的相似度，或者叫距离。

闵可夫斯基距离：

$$dist(X,Y) = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$$

当 p=1 时为曼哈顿距离。

当 p=2 时为欧氏距离。

当 p=∞时为切比雪夫距离。

KL 距离：

$$D(P||Q) = \sum_x P(x)\log(\frac{P(x)}{Q(x)})$$

Pearson 相关系数：

$$\rho_{XY} = \frac{\sum_{i=1}^{n}(X_i - \mu_X)(Y_i - \mu_Y)}{\sqrt{\sum_{i=1}^{n}(X_i - \mu_X)^2} * \sqrt{\sum_{i=1}^{n}(Y_i - \mu_Y)^2}}$$

$$dist(X,Y) = 1 - \rho_{XY}$$

传统 Kmeans 算法：假设输入样本为 X1，X2…Xm，设置 k 个初始类别中心 a1，a2…ak。需要注意 k 值的选择，一般根据先验经验选择一个合适的 k 值，如果没有什么先验经验，可以通过交叉验证选择一个合适的 k 值。质心的初始位置会很大程度上影响最终的结果，所以这些质心不能太近。对于每个样本，将其标记为距离（如欧氏距离）最近的类别中心的类别。全部分好后，更新每个类别的中心点 a 为隶属该类别的所有样本的均值。重复这个步骤直到达到某个中止条件。中止条件包括迭代次数、最小平方误差 MSE、簇中心点变化率等等。

收敛性：可以保证 Kmeans 算法是收敛的。设 J 函数是每个样本点到其质心距离的平方和。为了让 J 函数到达最小值，可以固定质心 u，调整每个样本点所属的类别 C，也可以固定 C，调整每个类别的质心 u。当 J 达到最小值时，C 和 u 也同时收敛。

但是由于 J 函数是非凸函数，不能保证极小值等于最小值。所以我们达到的极小值不一定是最优解。可以多次尝试这个算法，每次使用不同的初始值，然后选取所能够达到的最小值来作为最优解。

Sklearn 中的主要参数：n_clusters，即我们选取的 k 值，一般需要多试一些 k 值以达到最好的效果。**Max_iter**，最大迭代次数，我们使用的 J 函数不是凸函数，所以我们应该设置一个合适的迭代次数，以便算法退出循环。**N_init**，用不同的初始化质心运行算法的次数，由于 Kmeans 算法是结果受初始值影响的局部最优的迭代算法，所以需要多跑几次来选择一个最好的聚类效果，默认是 10，如果 k 值较大的话，可以适当增大这个值。**Init**，初始值选择的方式，random 为完全随机选择，k-means++为优化过的 Kmeans 算法。**Algorithm**，算法选择参数，full 是传统的 Kmeans 算法，elkan 是 elkan Kmeans 算法，auto 是算法根据数据值是否稀疏，决定使用 full 还是 elkan。

二分 Kmeans 算法：二分 Kmeans 算法是为了解决初始簇心比较敏感的问题，是一种弱化初始质心的算法。它将所有样本作为一个簇，放入一个队列中。每次从队列中选择一个簇进行 Kmeans 算法，划分为两个簇，再放入队列。循环直到达到中止条件。

选择簇的规则是对所有簇计算误差和 SSE，SSE 也可以认为是距离函数的一种变种，选择 SSE 最大的簇进行划分。

$$SSE = \sum_{i=1}^{n} w_i(y_i - \hat{y}_i)^2$$

Kmeans++算法：与传统 Kmeans 算法的区别在于初始 k 个质心的选择方面，Kmeans++算法采用以下步骤选定 k 个初始质点。任选一个点作为第一个聚类中心，对每个样本点计算它到所有聚类中心的距离和 D(x)，基于 D(x)采用线性概率选择下一个聚类中心点（距离较远的点成为新增的聚类中心点）。

$$D(x) = \arg\min \sum_{r=1}^{k} ||x_i - \mu_r||_2^2$$

Elkan Kmeans 距离计算优化算法：传统算法中，计算每个样本到质心的距离比较费时间，elkan Kmeans 算法是为了解决这个问题。原理是基于三角形两边之和大于第三边，两边之差小于第三边。例如，为了计算样本点 x 到 j1,j2 那个质心更近，如果

$$2D(x,j1) \leq D(j1,j2)$$

那么马上可以得出结论

$$D(x,j1) \leq D(x,j2)$$

就不需要在计算 D(x,j2)了，这样就省了一步计算。

另一个规律是，对于一个样本点 x 和两个质心 j1,j2

$$D(x,j2) \geq \max\{0, D(x,j1) - D(j1,j2)\}$$

利用这两个规律，elkan Kmeans 算法相比传统 Kmeans 算法速度有很大提高，但是如果样本特征是稀疏的，有缺失值的话，这个方法就不适用了。

大样本优化 mini Batch Kmeans 算法：这个算法使用样本集中的一部分样本来做传统的 Kmeans，这样可以避免巨大的计算量，算法收敛速度大大加快。代价也很明确，算法的精确度会降低，要让这个损失值在可以接受的范围内。

先从样本集中选取部分（batch size）数据，通过 Kmeans 算法建立 k 个质心，然后从样本集中不放回的抽取另一部分样本，加入到已有模型中，然后更新质心位置，重复抽取、更新，直到中心点稳定或者达到迭代次数，停止。

2、 代码实现

```
## Closure Functions
def tokenize(item):
    vector =
Vectors.dense(float(item[0]),float(item[1]),float(item[2]),float(item[3]
),float(item[4]),float(item[5]),float(item[7]),float(item[8]),float(item
[9]),float(item[10]),float(item[14]),float(item[15]),float(item[16]),flo
at(item[17]),float(item[18]),float(item[19]))
    if item[20] == "20":
        label = 0.0
    else:
        label = 1.0

    item = LabeledPoint(label,vector)
    return item

def vectored(item):
    vector =
Vectors.dense(float(item[0]),float(item[1]),float(item[2]),float(item[3]
),float(item[4]),float(item[5]),float(item[7]),float(item[8]),float(item
[9]),float(item[10]),float(item[14]),float(item[15]),float(item[16]),flo
at(item[17]),float(item[18]),float(item[19]))
    return vector


## Main functionality

def main(sc):
    iris_lines = sc.textFile("bank5.csv")
    #print(iris_lines.collect())

    iris_lines = iris_lines.map(lambda item: item.split(","))
    #print(iris_lines.collect())
 iris_vector = iris_lines.map(lambda item:vectored(item))
    print(iris_vector)
    summary = Statistics.colStats(iris_vector)
    max_vector = summary.max()
    min_vector = summary.min()
    print("max " + str(summary.max()))
```

```
    print("min " + str(summary.min()))
    def scale(item):
        item0 = float(item[0])
        item1 = float(item[1])
        item2 = float(item[2])
        item3 = float(item[3])
        item4 = float(item[4])
        item5 = float(item[5])
        item6 = float(item[6])
        ##item7 = float(item[7])
        item8 = float(item[8])
        item9 = float(item[9])
        item10 = float(item[10])
        item11 = float(item[11])
        ##item12 = float(item[12])
        ##item13 = float(item[13])
        ##item14 = float(item[14])
        item15 = float(item[15])
        item16 = float(item[16])
        item17 = float(item[17])
        item18 = float(item[18])
        item19 = float(item[19])
        fitem0 = (item0 - min_vector[0])/(max_vector[0] - min_vector[0])
        fitem1 = (item1 - min_vector[1])/(max_vector[1] - min_vector[1])
        fitem2 = (item2 - min_vector[2])/(max_vector[2] - min_vector[2])
        fitem3 = (item3 - min_vector[3])/(max_vector[3] - min_vector[3])
        fitem4 = (item4 - min_vector[4])/(max_vector[4] - min_vector[4])
        fitem5 = (item5 - min_vector[5])/(max_vector[5] - min_vector[5])
        fitem6 = (item6 - min_vector[6])/(max_vector[6] - min_vector[6])
        ##fitem7 = (item7 - min_vector[7])/(max_vector[7] -
min_vector[7])
        fitem8 = (item8 - min_vector[8])/(max_vector[8] - min_vector[8])
        fitem9 = (item9 - min_vector[9])/(max_vector[9] - min_vector[9])
        fitem10 = (item10 - min_vector[10])/(max_vector[10] -
min_vector[10])
        fitem11 = (item11 - min_vector[11])/(max_vector[11] -
min_vector[11])
        ##fitem12 = (item12 - min_vector[12])/(max_vector[12] -
min_vector[12])
        ##fitem13 = (item13 - min_vector[13])/(max_vector[13] -
min_vector[13])
        ##fitem14 = (item14 - min_vector[14])/(max_vector[14] -
min_vector[14])
```

```
        fitem15 = (item15 - min_vector[15])/(max_vector[15] -
min_vector[15])
        fitem16 = (item16 - min_vector[16])/(max_vector[16] -
min_vector[16])
        fitem17 = (item17 - min_vector[17])/(max_vector[17] -
min_vector[17])
        fitem18 = (item18 - min_vector[18])/(max_vector[18] -
min_vector[18])
        fitem19 = (item19 - min_vector[19])/(max_vector[19] -
min_vector[19])
        strval =
'%3.2f,%3.2f,%3.2f,%3.2f,%3.2f,%3.2f,%3.2f,%3.2f,%3.2f,%3.2f,%3
.2f,%3.2f,%3.2f,%3.2f,%3.2,%s' %
(fitem0,fitem1,fitem2,fitem3,fitem4,fitem5,fitem6,fitem8,fitem9,fitem
10,fitem11,fitem15,fitem16,fitem17,fitem18,fitem19, item[20])
        return strval
    iris_lines = iris_lines.map(lambda item:scale(item))

    iris_lines = iris_lines.map(lambda item: item.split(","))
    for iris_line_item in iris_lines.collect():
        print(iris_line_item)
    iris_vector = iris_lines.map(lambda item:vectored(item))

    #Build the model (cluster the data)

    model = KMeans.train(iris_vector, 2, maxIterations=1000,
initializationMode="random")
    # Evaluate clustering by computing Within Set Sum of Squared
Errors
    def error(point):
        center = model.centers[model.predict(point)]
        return sqrt(sum([x**2 for x in (point - center)]))

    #Evaluate clustering by computing Within Set Sum of Squared
Errors
    WSSSE = iris_vector.map(lambda point: error(point)).reduce(lambda
x, y: x + y)
    print("Within Set Sum of Squared Error = " + str(WSSSE))


    print("model.k:"+str(model.k))
    centers = model.clusterCenters
    print("Cluster Centers: ")
```

```
    for center in centers:
        print(center)

    iris_points = iris_lines.map(lambda item:tokenize(item))
    iris_compare= iris_points.map(lambda
item:(item.label,model.predict(item.features)))
    for iris_comp_item in iris_compare.collect():
        print(iris_comp_item)

    sc.stop()


if __name__ == "__main__":
    # Configure Spark
    conf = SparkConf().setAppName(APP_NAME)
    conf = conf.setMaster("local[*]")
    sc   = SparkContext(conf=conf)
    # Execute Main functionality
    main(sc)
```

Sc.textFile()函数读取 bank5.csv 文件。

Item.split()用逗号分隔文件，形成一个二维数组 iris_lines。

Vectored(item)函数把 iris_lines 变成一个向量数组。

Statistics.colStats()函数计算以列为基础的计算统计量的基本数据。得到变量 summary。再得到 max=summary.max()和 min=summary.min()。打印 summary 可以看到数据集的基本信息。

定义一个 Scale()函数，综合计算每个特征值。通过这个结果，计算得到一个向量 iris_vector，用来作为 Kmeans 算法的输入。

Kmeans.train()函数是 Kmeans 算法的核心函数。将 iris_vector 作为参数传进去，k 的数量为 2，这是人为设定的，maxIterations 最大迭代次数是 100，这是默认值。initializationMode 改成了 random，默认是 k-means||。用这个函数得到了 model 模型。

下面计算了 WSSSE（Within Set Sum of Squared Error），使用自己定义的 error()函数，计算样本点到质心的距离。

质心的结果用 model.clusterCenters 打印出来，k 值用 model.k 打印出来。

六、 感受

参加此次科研营，收获最多的是机器学习算法的掌握。从最初的 KNN 算法，再到更高级的逻辑回归算法、朴素贝叶斯、神经网络，这些算法是机器学习的算法核心，虽然 6 天的时间不足以全部掌握，但是有一个粗浅的了解对日后机器学习的深入研究也起到入门的作用。

此外，我们使用 pyspark 库或者 sklearn 库来写机器学习的程序，这些库都是非常好用的完全封装的函数，我们只需要知道参数的意义，然后调用函数就能得到可移植的模型 model。对我们来说，机器学习的算法只要能够了解就足够了，如何使用现有的库函数去完成某个工作才是应该考虑的。经过许多年的发展，这些库函数已经写的很完备，我们不必要自己再去写代码实现算法。

在助教的帮助下，我们了解了文本分析——机器学习的最基本问题。古代笑话和现代笑话的分类很能说明问题，这就是一个用 pyspark 库完成的机器学习的例子。用合适的分类器，选出合适的特征，然后用数据集训练成模型，就是一个可以做预测的机器学习实例。

除此之外，我们小组的组内气氛非常好，六个人从第一天起就混的很熟，可能是我们的性情彼此相投。在每天晚上一起完成任务的时候，时间也是在轻松愉快中度过的。最后一天课程结束的日子，我们一起去外面吃饭，给每个人送行。我相信我们小组六个人在今后留学的道路上还会始终保持联系，互相帮助。

总之，这次科研营的收获很多，我的所见所闻将会影响我未来的道路，以及日后研究方向的选择。