# 机器学习算法总结2

## 6. 朴素贝叶斯

参考文章：

- 《统计学习方法》
- 机器学习常见算法个人总结（面试用）
- 朴素贝叶斯理论推导与三种常见模型
- 朴素贝叶斯的三个常用模型：高斯、多项式、伯努利

### 简介

> 朴素贝叶斯是基于贝叶斯定理与特征条件独立假设的分类方法。

贝叶斯定理是基于条件概率来计算的，条件概率是在已知事件B发生的前提下，求解事件A发生的概率，即$P(A|B)=\frac{P(AB)}{P(B)}$，而贝叶斯定理则可以通过$P(A|B)$来求解$P(B|A)$：

$$P(B|A)=\frac{P(A|B)P(B)}{P(A)}$$

其中分母$P(A)$可以根据全概率公式分解为：$P(A)=\sum_{i=1}^{n}P(B_i)P(A|B_i)$

而特征条件独立假设是指假设各个维度的特征$x_1,x_2,\ldots,x_n$互相独立，则条件概率可以转化为：

$$P(x|y_k)=P(x_1,x_2,\ldots,x_n|y_k)=\prod_{i=1}^{n}P(x_i|y_k)$$

朴素贝叶斯分类器可表示为：

$$f(x)=argmax_{y_k}P(y_k|x)=argmax_{y_k}\frac{P(y_k)\prod_{i=1}^{n}P(x_i|y_k)}{\sum_k P(y_k)\prod_{i=1}^{n}P(x_i|y_k)}$$

而由于对上述公式中分母的值都是一样的，所以可以忽略分母部分，即可以表示为：

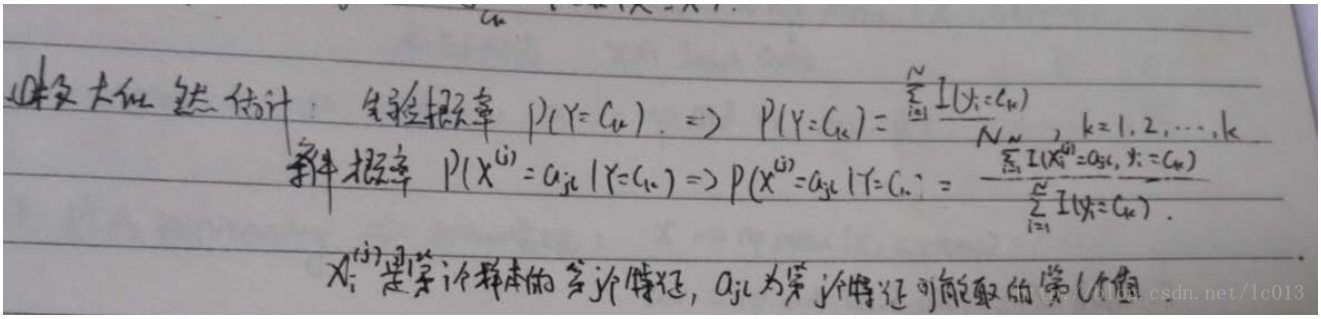
$$f(x)=argmaxP(y_k)\prod_{i=1}^{n}P(x_i|y_k)$$

这里$P(y_k)$是先验概率，而$P(y_k|x)$则是后验概率，朴素贝叶斯的目标就是最大化后验概率，这等价于期望风险最小化。

### 参数估计

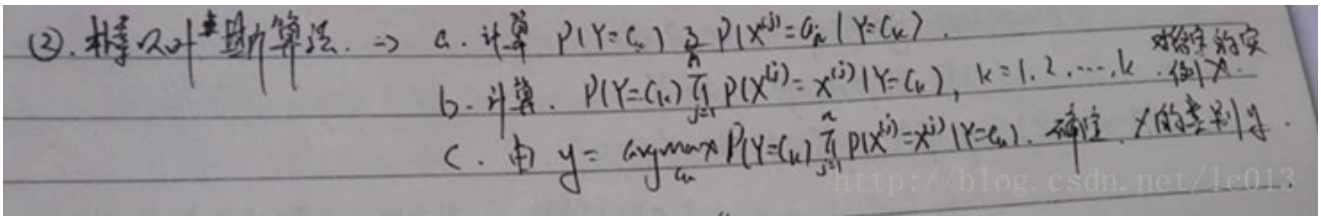
#### 极大似然估计

朴素贝叶斯的学习意味着估计$P(y_k)$和$P(x_i|y_k)$,可以通过极大似然估计来估计相应的概率。




如上图所示，分别是$P(y_k)$和$P(x_i|y_k)$的极大似然估计。

当求解完上述两个概率，就可以对测试样本使用朴素贝叶斯分类算法来预测其所属于的类别，简单总结的算法流程如下所示：




## 贝叶斯估计/多项式模型

用极大似然估计可能会出现所要估计的概率值为**0**的情况，这会影响到后验概率的计算，使分类产生偏差。解决这个问题的办法是使用贝叶斯估计，也被称为多项式模型。

当特征是离散的时候，使用多项式模型。多项式模型在计算先验概率$P(y_k)$和条件概率$P(x_i|y_k)$时，会做一些平滑处理，具体公式为：

$$P(y_k)=\frac{N_{y_k}+\alpha}{N+k\alpha}$$

> $N$是总的样本个数，$k$是总的类别个数，$N_{y_k}$是类别为$y_k$的样本个数，$\alpha$是平滑值。

$$P(x_i|y_k)=\frac{N_{y_k,x_i}+\alpha}{N_{y_k}+n\alpha}$$

> $N_{y_k}$是类别为$y_k$的样本个数，$n$是特征的维数，$N_{y_k,x_i}$是类别为$y_k$的样本中，第$i$维特征的值是$x_i$的样本个数，$\alpha$是平滑值。

当$\alpha=1$时，称作**Laplace**平滑，当$0<\alpha<1$时，称作**Lidstone**平滑，$\alpha=0$时不做平滑。

如果不做平滑，当某一维特征的值$x_i$没在训练样本中出现过时，会导致$P(x_i|y_k)=0$，从而导致后验概率为0。加上平滑就可以克服这个问题。

## 高斯模型

当特征是连续变量的时候，运用多项式模型会导致很多$P(x_i|y_k)=0$（不做平滑的情况下），即使做平滑，所得到的条件概率也难以描述真实情况，所以处理连续变量，应该采用高斯模型。

高斯模型是假设每一维特征都服从高斯分布（正态分布）：

$$P(x_i|y_k)=\frac{1}{\sqrt{2\pi\sigma_{y_k}^2}}exp(-\frac{(x_i-\mu_{y_k})^2}{2\sigma_{y_k}^2})$$

$\mu_{y_k,i}$表示类别为$y_k$的样本中，第$i$维特征的均值；
$\sigma^2_{y_k,i}$表示类别为$y_k$的样本中，第$i$维特征的方差。

### 伯努利模型

与多项式模型一样，伯努利模型适用于离散特征的情况，所不同的是，伯努利模型中每个特征的取值只能是**1**和**0**(以文本分类为例，某个单词在文档中出现过，则其特征值为1，否则为0).

伯努利模型中，条件概率$P(x_i|y_k)$的计算方式是：

当特征值$x_i$为1时，$P(x_i|y_k)=P(x_i=1|y_k)$；

当特征值$x_i$为0时，$P(x_i|y_k)=1-P(x_i=1|y_k)$；

## 工作流程

1. 准备阶段
   确定特征属性，并对每个特征属性进行适当划分，然后由人工对一部分待分类项进行分类，形成训练样本。
2. 训练阶段
   计算每个类别在训练样本中的出现频率及每个特征属性划分对每个类别的条件概率估计
3. 应用阶段
   使用分类器进行分类，输入是分类器和待分类样本，输出是样本属于的分类类别

## 属性特征

1. 特征为离散值时直接统计即可（表示统计概率）
2. 特征为连续值的时候假定特征符合高斯分布，则有

$$P(x_i|y_k)=\frac{1}{\sqrt{2\pi\sigma^2_{y_k}}}exp(-\frac{(x_i-\mu_{y_k})^2}{2\sigma^2_{y_k}})$$

## 与逻辑回归的不同

1. **Naive Bayes**是一个生成模型，在计算P(y|x)之前，先要从训练数据中计算P(x|y)和P(y)的概率，从而利用贝叶斯公式计算P(y|x)。

   **Logistic Regression**是一个判别模型，它通过在训练数据集上最大化判别函数P(y|x)学习得到，不需要知道P(x|y)和P(y)。

2. Naive Bayes是建立在条件独立假设基础之上的，设特征X含有n个特征属性（X1，X2，...Xn），那么在给定Y的情况下，X1，X2，...Xn是条件独立的。

   Logistic Regression的限制则要宽松很多，如果数据满足条件独立假设，Logistic Regression能够取得非常好的效果；当数据不满足条件独立假设时，Logistic Regression仍然能够通过调整参数让模型最大化的符合数据的分布，从而训练得到在现有数据集下的一个最优模型。

3. 当数据集比较小的时候，应该选用**Naive Bayes**，为了能够取得很好的效果，数据的需求量为O(log n)

当数据集比较大的时候，应该选用**Logistic Regression**，为了能够取得很好的效果，数据的需求量为O( n)

## 与逻辑回归的相同

1. 两者都是对特征的线性表达
2. 两者建模的都是条件概率，对最终求得的分类结果有很好的解释性。

# 优缺点

## 优点

1. 对小规模的数据表现很好，适合多分类任务，适合增量式训练。

## 缺点

1. 对输入数据的表达形式很敏感（离散、连续，值极大极小之类的）。

# 代码实现

下面是使用`sklearn`的代码例子，分别实现上述三种模型,例子来自朴素贝叶斯的三个常用模型：高斯、多项式、伯努利。
下面是高斯模型的实现

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris.feature_names  # 四个特征的名字
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
>>> iris.data
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2],
       [ 5.4,  3.9,  1.7,  0.4],
       [ 4.6,  3.4,  1.4,  0.3],
       [ 5. ,  3.4,  1.5,  0.2],
       ......
       [ 6.5,  3. ,  5.2,  2. ],
       [ 6.2,  3.4,  5.4,  2.3],
       [ 5.9,  3. ,  5.1,  1.8]]) #类型是numpy.array
>>> iris.data.size
600  #共600/4=150个样本
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'],
      dtype='|S10')
>>> iris.target
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,....., 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, ......, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
>>> iris.target.size
150
>>> from sklearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(iris.data, iris.target)
>>> clf.predict(iris.data[0])
array([0])   # 预测正确
>>> clf.predict(iris.data[149])
array([2])   # 预测正确
>>> data = numpy.array([6,4,6,2])
>>> clf.predict(data)
array([2])  # 预测结果很合理
```

多项式模型如下：

```
>>> import numpy as np
>>> X = np.random.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.fit(X, y)
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
>>> print(clf.predict(X[2]))
[3]
```

值得注意的是，多项式模型在训练一个数据集结束后可以继续训练其他数据集而无需将两个数据集放在一起进行训练。在sklearn中，MultinomialNB()类的partial_fit()方法可以进行这种训练。这种方式特别适合于训练集大到内存无法一次性放入的情况。

在第一次调用`partial_fit()`时需要给出所有的分类标号。

```
>>> import numpy
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.partial_fit(numpy.array([1,1]), numpy.array(['aa']), ['aa','bb'])
GaussianNB()
>>> clf.partial_fit(numpy.array([6,1]), numpy.array(['bb']))
GaussianNB()
>>> clf.predict(numpy.array([9,1]))
array(['bb'],
      dtype='|S2')
```

伯努利模型如下：

```
>>> import numpy as np
>>> X = np.random.randint(2, size=(6, 100))
>>> Y = np.array([1, 2, 3, 4, 4, 5])
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(X, Y)
BernoulliNB(alpha=1.0, binarize=0.0, class_prior=None, fit_prior=True)
>>> print(clf.predict(X[2]))
[3]
```

朴素贝叶斯的总结就到这里为止。

# 7. K-近邻算法(KNN)

## 简介

> k近邻（KNN)是一种基本分类与回归方法。

其思路如下：给一个训练数据集和一个新的实例，在训练数据集中找出与这个新实例最近的$k$个训练实例，然后统计最近的$k$个训练实例中所属类别计数最多的那个类，就是新实例的类。其流程如下所示：

1. 计算训练样本和测试样本中每个样本点的距离（常见的距离度量有欧式距离，马氏距离等）；
2. 对上面所有的距离值进行排序；
3. 选前$k$个最小距离的样本；
4. 根据这$k$个样本的标签进行投票，得到最后的分类类别；

KNN的特殊情况是$k=1$的情况，称为最近邻算法。对输入的实例点（特征向量）$x$，最近邻法将训练数据集中与$x$最近邻点的类作为其类别。

## 三要素

1. $k$值的选择
2. 距离的度量（常见的距离度量有欧式距离，马氏距离）

3. 分类决策规则（多数表决规则）

## k值的选择

1. $k$值越小表明模型越复杂，更加容易过拟合，其偏差小，而方差大
2. 但是$k$值越大，模型越简单，如果$k=N$的时候就表明无论什么点都是训练集中类别最多的那个类，这种情况，则是偏差大，方差小。

> 所以一般$k$会取一个较小的值，然后用过交叉验证来确定
> 这里所谓的交叉验证就是将样本划分一部分出来为预测样本，比如95%训练，5%预测，然后$k$分别取1，2，3，4，5之类的，进行预测，计算最后的分类误差，选择误差最小的$k$

## 距离的度量

KNN算法使用的距离一般是欧式距离，也可以是更一般的$L_p$距离或者马氏距离，其中$L_p$距离定义如下：

$$L_p(x_i,x_j)=(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^p)^{\frac{1}{p}}$$

这里$x_i=(x_i^{(1)},x_i^{(2)},...x_i^{(n)})^T,x_j=(x_j^{(1)},x_j^{(2)},...,x_j^{(n)})^T$，然后$p\geq 1$。

当$p=2$，称为欧式距离，即

$$L_2(x_i,x_j)=(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^2)^{\frac{1}{2}}$$

当$p=1$，称为曼哈顿距离，即

$$L_1(x_i,x_j)=\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|$$

当$p=\infty$，它是各个坐标距离的最大值，即

$$L_\infty(x_i,x_j)=max_l|x_i^{(l)}-x_j^{(l)}|$$

马氏距离如下定义：

**马氏距离**是由印度统计学家马哈拉诺比斯（P. C. Mahalanobis）提出的，表示数据的协方差距离。它是一种有效的计算两个未知样本集的相似度的方法。与欧氏距离不同的是它考虑到各种特性之间的联系（例如：一条关于身高的信息会带来一条关于体重的信息，因为两者是有关联的）并且是尺度无关的（scale-invariant），即独立于测量尺度。 对于一个均值为$\mu=(\mu_1,\mu_2,\mu_3,...,\mu_p)^T$，协方差矩阵为$\sum$的多变量向量$x=(x_1,x_2,x_3,...,x_p)^T$，其马氏距离为

$$D_M(x)=\sqrt{(x-\mu)^T\Sigma^{-1}(x-\mu)}$$

**马氏距离**也可以定义为两个服从同一分布并且其协方差矩阵为$\sum$的随机变量$\vec{x}$与$\vec{y}$的差异程度：

$$d(\vec{x},\vec{y})=\sqrt{(\vec{x}-\vec{y})^T\Sigma^{-1}(\vec{x}-\vec{y})}$$

如果协方差矩阵为单位矩阵，马氏距离就简化为欧氏距离；如果协方差矩阵为对角阵，其也可称为*正规化的欧氏距离*。

$$d(\vec{x},\vec{y})=\sqrt{\sum_{i=1}^{p}\frac{(x_i-y_i)^2}{\sigma_i^2}}$$

其中$\sigma_i$是$x_i$的标准差。



## KNN的回归

在找到最近的$k$个实例之后，可以计算这$k$个实例的平均值作为预测值。或者还可以给这$k$个实例添加一个权重再求平均值，这个权重与度量距离成反比（越近权重越大）。

## 优缺点

### 优点

1. 思想简单，理论成熟，既可以用来做分类也可以用来做回归；
2. 可用于非线性分类；
3. 训练时间复杂度为$O(n)$；
4. 准确度高，对数据没有假设，对异常值不敏感；

### 缺点

1. 计算量大；
2. 样本不平衡问题（即有些类别的样本数量很多，而其它样本的数量很少）；
3. 需要大量的内存；

## KD树

> KD树是一个二叉树，表示对K维空间的一个划分，可以进行快速检索（那KNN计算的时候不需要对全样本进行距离的计算了）

## 构造KD树

在k维的空间上循环找子区域的中位数进行划分的过程。
假设现在有K维空间的数据集$T = x_1, x_2, x_3, ... x_n, x_i = a_1, a_2, a_3 .. a_k$

1. 首先构造根节点，以坐标$a_1$的中位数$b$为切分点，将根结点对应的矩形局域划分为两个区域，区域1中$a_1 < b$, 区域2中$a_1 > b$
2. 构造叶子节点，分别以上面两个区域中$a_2$的中位数作为切分点，再次将他们两两划分，作为深度1的叶子节点，（如果$a_2 =$ 中位数，则$a_2$的实例落在切分面）
3. 不断重复2的操作，深度为$j$的叶子节点划分的时候，选择维度为$l = j(modk) + 1$，索取的$a_i$ 的$i = j$，直到两个子区域没有实例时停止

## KD树的搜索

1. 首先从根节点开始递归往下找到包含$x$的叶子节点，每一层都是找对应的$x_i$
2. 将这个叶子节点认为是当前的“近似最近点”
3. 递归向上回退，如果以$x$圆心，以“近似最近点”为半径的球与根节点的另一半子区域边界相交，则说明另一半子区域中存在与$x$更近的点，则进入另一个子区域中查找该点并且更新”近似最近点“
4. 重复3的步骤，直到另一子区域与球体不相交或者退回根节点
5. 最后更新的”近似最近点“与$x$真正的最近点

## KD树进行KNN查找

通过KD树的搜索找到与搜索目标最近的点，这样KNN的搜索就可以被限制在空间的局部区域上了，可以大大增加效率。

## KD树搜索的复杂度

当实例随机分布的时候，搜索的复杂度为$log(N)$，$N$为实例的个数，KD树更加适用于实例数量远大于空间维度的KNN搜索，如果实例的空间维度与实例个数差不多时，它的效率基于等于线性扫描。

## 代码实现

使用`sklearn`的简单代码例子：

```
#Import Library
from sklearn.neighbors import KNeighborsClassifier

#Assumed you have, X (predictor) and Y (target) for training data set and x_test(predictor) of
test_dataset
# Create KNeighbors classifier object model

KNeighborsClassifier(n_neighbors=6) # default value for n_neighbors is 5

# Train the model using the training sets and check score
model.fit(X, y)

#Predict Output
predicted= model.predict(x_test)
```

最后，在用**KNN**前你需要考虑到：

- KNN的计算成本很高
- 所有特征应该标准化数量级，否则数量级大的特征在计算距离上会有偏移。
- 在进行KNN前预处理数据，例如去除异常值，噪音等。

# 8 K-均值算法

参考自：

- 《机器学习》
- 机器学习&数据挖掘笔记_16（常见面试之机器学习算法思想简单梳理）
- K-Means Clustering
- 斯坦福大学公开课：机器学习课程

## 简介

> K-均值是最普及的聚类算法，算法接受一个未标记的数据集，然后将数据集聚类成不同的组。

K-均值是一个迭代算法，假设我们想要将数据聚类成n个组，其方法为：

1. 首先选择**K**个随机的点，称其为聚类中心
2. 对于数据集中的每一个数据，按照距离**K**个中心点的距离，将其与距离最近的中心点关联起来，与同一个中心点关联的所有点聚成一个类
3. 计算每一个组的平均值，将该组所关联的中心点移动到平均值的位置
4. 重复步骤2-3，直到中心点不再变化

这个过程中分两个主要步骤，第一个就是第二步，将训练集中的样本点根据其与聚类中心的距离，分配到距离最近的聚类中心处，接着第二个就是第三步，更新类中心，做法是计算每个类的所有样本的平均值，然后将这个平均值作为新的类中心值，接着继续这两个步骤，直到达到终止条件，一般是指达到设定好的迭代次数。

当然在这个过程中可能遇到有聚类中心是没有分配数据点给它的，通常的一个做法是删除这种聚类中心，或者是重新选择聚类中心，保证聚类中心数还是初始设定的**K**个。

# 优化目标

K-均值最小化问题，就是最小化所有的数据点与其所关联的聚类中心之间的距离之和，因此K-均值的代价函数（又称为畸变函数）为：

$$J(c^{(1)},c^{(2)},\dots,c^{(m)},\mu_1,\mu_2,\dots,\mu_m)=\frac{1}{m}\sum_{i=1}^{m}||x^(i)-\mu_{c(i)}||^2$$

其中$\mu_{c(i)}$代表与$x^{(i)}$最近的聚类中心点。

所以我们的优化目标是找出是的代价函数最小的$c^{(1)},c^{(2)},\dots,c^{(m)}$和$\mu_1,\mu_2,\dots,\mu_m$:

$$min_{c^{(1)},c^{(2)},\dots,c^{(m)},\mu_1,\mu_2,\dots,\mu_m}J(c^{(1)},c^{(2)},\dots,c^{(m)},\mu_1,\mu_2,\dots,\mu_m)$$

回顾K-均值迭代算法的过程可知，第一个循环就是用于减小$c^{(i)}$引起的代价，而第二个循环则是用于减小$\mu_i$引起的代价，因此，迭代的过程一定会是每一次迭代都在减小代价函数，不然便是出现了错误。

# 随机初始化

在运行K-均值算法之前，首先需要随机初始化所有的聚类中心点，做法如下：

1. 首先应该选择$K<m$,即聚类中心点的个数要小于所有训练集实例的数量
2. 随机选择$K$个训练实例，然后令$K$个聚类中心分别于这K个训练实例相等

K-均值的一个问题在于，它有可能会停留在一个局部最小值处，而这取决于初始化的情况。

为了解决这个问题，通常需要多次运行**K-**均值算法，每一次都重新进行随机初始化，最后再比较多次运行**K-**均值的结果，选择代价函数最小的结果。这种方法在**K**较小（**2-10**）的时候还是可行的，但是如果K较大，这种做法可能不会有明显地改善。

# 优缺点

## 优点

1. k-means算法是解决聚类问题的一种经典算法，算法简单、快速。
2. 对处理大数据集，该算法是相对可伸缩的和高效率的，因为它的复杂度大约是$O(nkt)$，其中$n$是所有对象的数目，$k$是簇的数目,$t$是迭代的次数。通常$k<<n$。这个算法通常局部收敛。
3. 算法尝试找出使平方误差函数值最小的**k**个划分。当簇是密集的、球状或团状的，且簇与簇之间区别明显时，聚类效果较好。

## 缺点

1. k-平均方法只有在簇的平均值被定义的情况下才能使用，且对有些分类属性的数据不适合。
2. 要求用户必须事先给出要生成的簇的数目$k$。
3. 对初值敏感，对于不同的初始值，可能会导致不同的聚类结果。
4. 不适合于发现非凸面形状的簇，或者大小差别很大的簇。
5. 对于"噪声"和孤立点数据敏感，少量的该类数据能够对平均值产生极大影响。

# 代码实现

代码参考自K-Means Clustering。

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
@Time    : 2016/10/21 16:35
@Author  : cai

实现 K-Means 聚类算法
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
import os

# 寻址最近的中心点
def find_closest_centroids(X, centroids):
    m = X.shape[0]
    k = centroids.shape[0]
    idx = np.zeros(m)

    for i in range(m):
        min_dist = 1000000
        for j in range(k):
            # 计算每个训练样本和中心点的距离
            dist = np.sum((X[i, :] - centroids[j, :]) ** 2)
            if dist < min_dist:
                # 记录当前最短距离和其中心的索引值
                min_dist = dist
                idx[i] = j

    return idx

# 计算聚类中心
def compute_centroids(X, idx, k):
    m, n = X.shape
    centroids = np.zeros((k, n))

    for i in range(k):
        indices = np.where(idx == i)
        # 计算下一个聚类中心，这里简单的将该类中心的所有数值求平均值作为新的类中心
        centroids[i, :] = (np.sum(X[indices, :], axis=1) / len(indices[0])).ravel()

    return centroids

# 初始化聚类中心
def init_centroids(X, k):
    m, n = X.shape
    centroids = np.zeros((k, n))
    # 随机初始化 k 个 [0,m]的整数
    idx = np.random.randint(0, m, k)

    for i in range(k):

        centroids[i, :] = X[idx[i], :]
```

```
    return centroids

# 实现 kmeans 算法
def run_k_means(X, initial_centroids, max_iters):
    m, n = X.shape
    # 聚类中心的数目
    k = initial_centroids.shape[0]
    idx = np.zeros(m)
    centroids = initial_centroids

    for i in range(max_iters):
        idx = find_closest_centroids(X, centroids)
        centroids = compute_centroids(X, idx, k)

    return idx, centroids

dataPath = os.path.join('data', 'ex7data2.mat')
data = loadmat(dataPath)
X = data['X']

initial_centroids = init_centroids(X, 3)
# print(initial_centroids)
# idx = find_closest_centroids(X, initial_centroids)
# print(idx)

# print(compute_centroids(X, idx, 3))

idx, centroids = run_k_means(X, initial_centroids, 10)
# 可视化聚类结果
cluster1 = X[np.where(idx == 0)[0], :]
cluster2 = X[np.where(idx == 1)[0], :]
cluster3 = X[np.where(idx == 2)[0], :]

fig, ax = plt.subplots(figsize=(12, 8))
ax.scatter(cluster1[:, 0], cluster1[:, 1], s=30, color='r', label='Cluster 1')
ax.scatter(cluster2[:, 0], cluster2[:, 1], s=30, color='g', label='Cluster 2')
ax.scatter(cluster3[:, 0], cluster3[:, 1], s=30, color='b', label='Cluster 3')
ax.legend()
plt.show()

# 载入一张测试图片，进行测试
imageDataPath = os.path.join('data', 'bird_small.mat')
image = loadmat(imageDataPath)
# print(image)

A = image['A']
print(A.shape)

# 对图片进行归一化
A = A / 255.


# 重新调整数组的尺寸
```

```
X = np.reshape(A, (A.shape[0] * A.shape[1], A.shape[2]))
# 随机初始化聚类中心
initial_centroids = init_centroids(X, 16)
# 运行聚类算法
idx, centroids = run_k_means(X, initial_centroids, 10)

# 得到最后一次的最近中心点
idx = find_closest_centroids(X, centroids)
# map each pixel to the centroid value
X_recovered = centroids[idx.astype(int), :]
# reshape to the original dimensions
X_recovered = np.reshape(X_recovered, (A.shape[0], A.shape[1], A.shape[2]))

# plt.imshow(X_recovered)
# plt.show()
```

完整代码例子和数据可以查看Kmeans练习代码。

# 9 提升方法

参考自：

- 《统计学习方法》
- 浅谈机器学习基础（上）
- Ensemble learning:Bagging,Random Forest,Boosting

## 简介

> 提升方法(boosting)是一种常用的统计学习方法，在分类问题中，它通过改变训练样本的权重，学习多个分类器，并将这些分类器进行线性组合，提供分类的性能。

## boosting和bagging

**boosting**和**bagging**都是集成学习（**ensemble learning**）领域的基本算法，**boosting**和**bagging**使用的多个分类器的类型是一致的。

### Bagging

**bagging**也叫自助汇聚法（**bootstrap aggregating**），比如原数据集中有N个样本，我们每次从原数据集中有放回的抽取，抽取N次，就得到了一个新的有N个样本的数据集，然后我们抽取S个N次，就得到了S个有N个样本的新数据集，然后拿这S个数据集去训练S个分类器，之后应用这S个分类器进行分类，选择分类器投票最多的类别作为最后的分类结果。一般来说自助样本的包含有63%的原始训练数据，因为：

假设共抽取$N$个样本，则$N$次都没有抽到的概率是$p = (1 - \frac{1}{N})^N$

则一个样本被抽到的概率有$p = 1 - (1 - \frac{1}{N})^N$

所以，当$N$很大时有：$p = 1 - \frac{1}{e} = 0.632$。

这样，在一次bootstrap的过程中，会有36%的样本没有被采样到，它们被称为**out-off-bag(oob)**，这是自助采样带给`bagging`的里一个优点，因为我们可以用`oob`进行“包外估计”**out-of-bag estimate**。

**bagging**通过降低基分类器的方差改善了泛化误差，**bagging**的性能依赖于基分类器的稳定性。如果基分类器是不稳定的，bagging有助于减少训练数据的随机波动导致的误差，如果基分类器是稳定的，即对训练数据集中的微小变化是鲁棒的，则组合分类器的误差主要由基分类器偏移所引起的，这种情况下，**bagging**可能不会对基分类器有明显的改进效果，甚至可能降低分类器的性能。

### **boosting**与**bagging**的区别

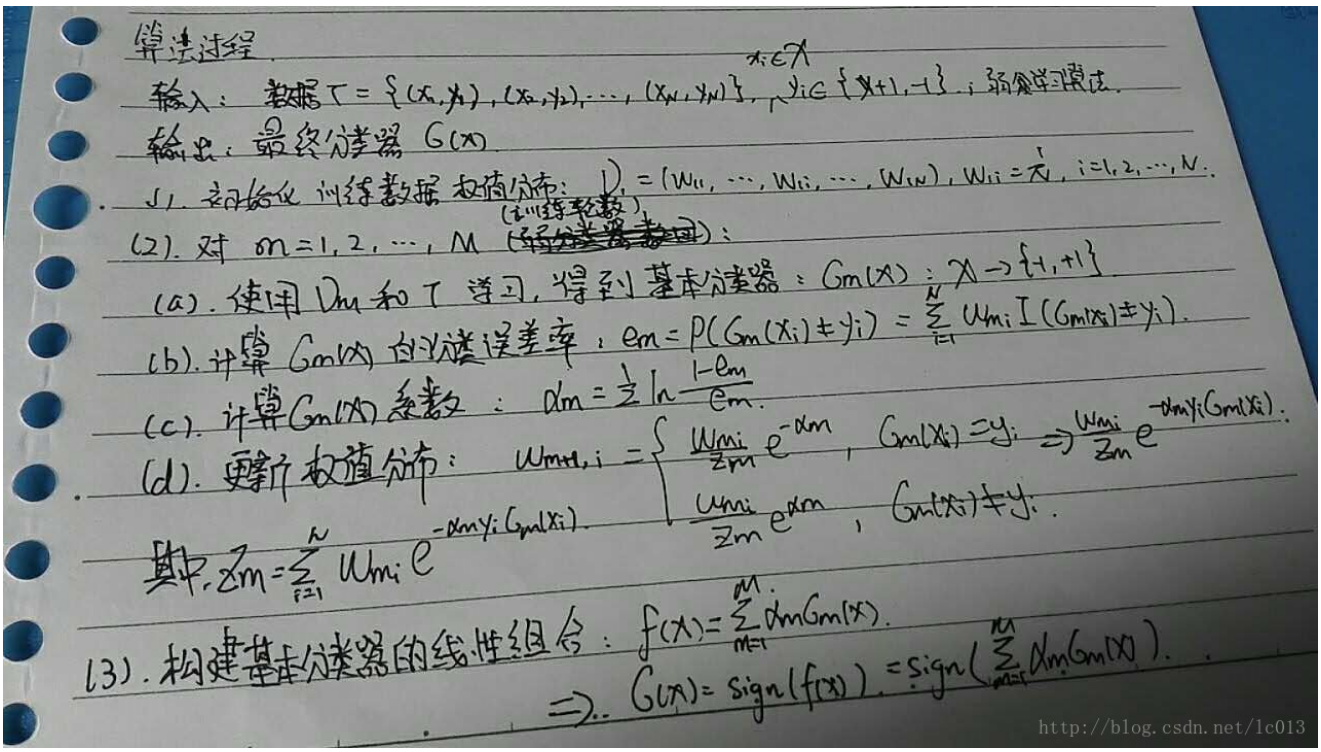
- bagging通过有放回的抽取得到了S个数据集，而boosting用的始终是原数据集，但是样本的权重会发生改变。
- **boosting**对分类器的训练是串行的，每个新分类器的训练都会受到上一个分类器分类结果的影响。
- **bagging**里面各个分类器的权重是相等的，但是**boosting**不是，每个分类器的权重代表的是其对应分类器在上一轮分类中的成功度。

**AdaBoost**是**boosting**方法中最流行的版本

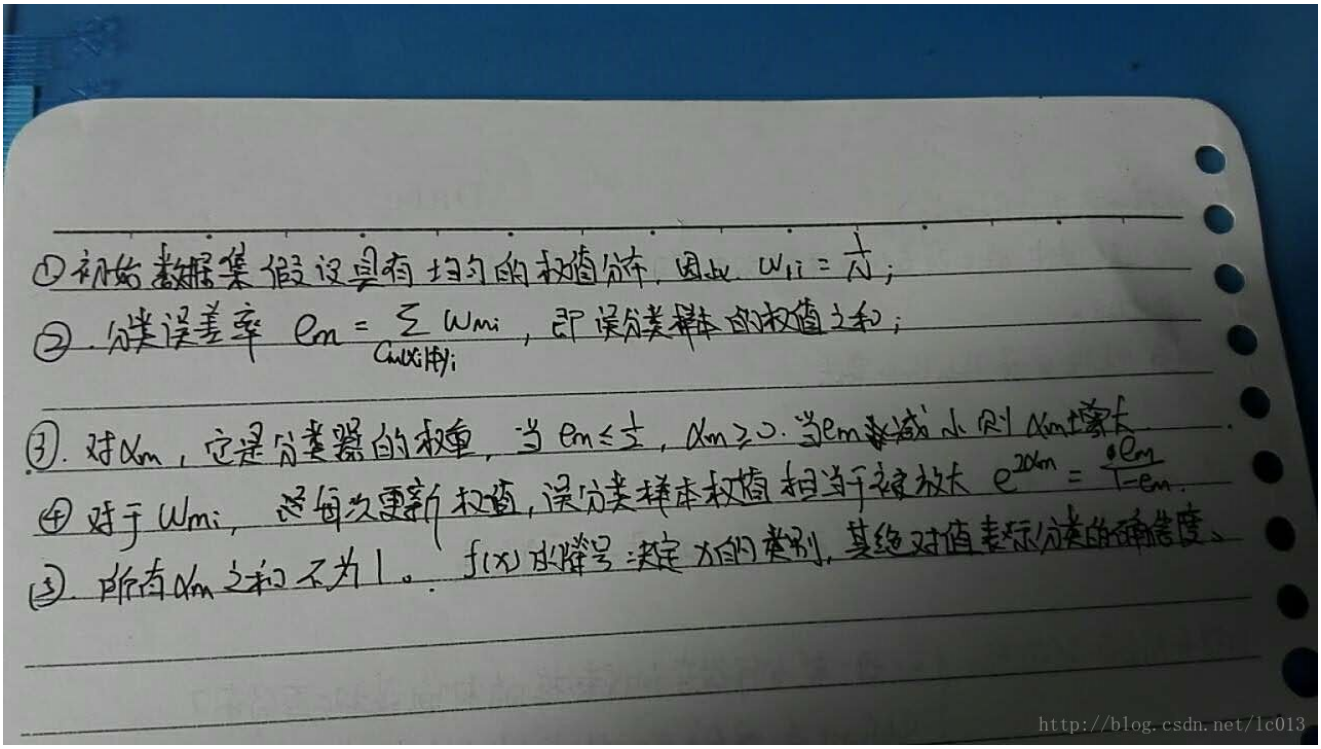
## AdaBoosts算法

> **AdaBoost**（**adaptive boosting**）是元算法，通过组合多个弱分类器来构建一个强分类器。我们为训练数据中的每一个样本都赋予其一个权重，这些权重构成了向量$\boldsymbol{D}$，一开始，这些权重都初始化成相等值，然后每次添加一个弱分类器对样本进行分类，从第二次分类开始，将上一次分错的样本的权重提高，分对的样本权重降低，持续迭代。此外，对于每个弱分类器而言，每个分类器也有自己的权重，取决于它分类的加权错误率，加权错误率越低，则这个分类器的权重值$\alpha$越高，最后综合多个弱分类器的分类结果和其对应的权重$\alpha$得到预测结果，AdaBoost是最好的监督学习分类方法之一。

其算法过程如下所示：




其中，注意：




## 训练误差分析

**AdaBoost**算法的最基本性质是在学习过程中不断减小训练误差，对训练误差的上界有如下定理：

> 定理1：AdaBoost最终分类器的训练误差界为：
>
> $$\frac{1}{N}\sum_{i=1}^{N} I(G(x_i)\neq y_i)\leq \frac{1}{N}\sum_{i} exp(-y_i f(x_i))=\prod_{m} Z_m$$
>
> 定理2：二类分类问题
>
> $$\prod_{m=1}^{M} Z_m=\prod_{m=1}^{M}[2\sqrt{e_m(1-e_m)}]=\prod_{m=1}^{M}[\sqrt{1-4\gamma_m^2}\leq exp(-2\sum_{m=1}^{M}\gamma_m^2)$$

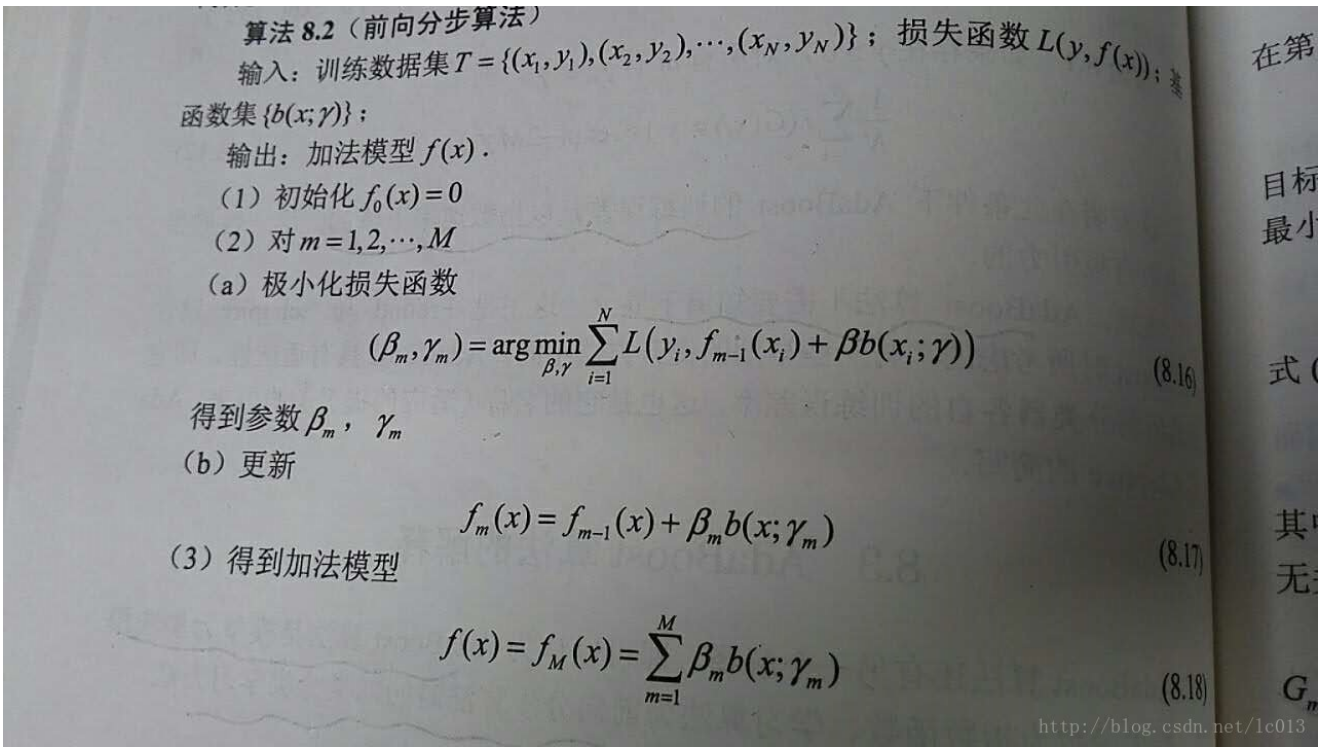
## 算法解释

**AdaBoost**算法还可以解释为模型是加法模型，损失函数是指数函数，学习算法是前向分步算法的二类分类学习方法。

加法模型是形如$f(x)=\sum_{i=1}^{M}\beta_m b(x;\gamma_m)$的函数形式，其中$b(x;\gamma_m)$是基函数，而$\beta_m$是基函数的系数，$\gamma_m$是基函数的参数。对于**AdaBoost**算法，其基本分类器的线性组合为$f(x)=\sum_{m=1}^{M}\alpha_m G_m(x)$正是一个加法模型。

**AdaBoost**算法的损失函数是指数函数，公式为$E=\sum_{i=1}^{N} exp(-y_i G_m(x_i))$。

此外，经过$m$轮迭代可以得到$f_m(x)=f_{m-1}(x)+\alpha_m G_m(x)$。而前向分步算法的过程如下所示：




通过上述步骤，前向分步算法将同时求解从$m=1$到$M$所有参数$\beta_m,\gamma_m$的优化问题简化为逐步求解各个$\beta_m,\gamma_m$的优化问题。

## 优缺点

### 优点

1. 泛化误差低
2. 容易实现，分类准确率较高，没有太多参数可以调

### 缺点

- 对异常值比较敏感
- 训练时间过长
- 执行效果依赖于弱分类器的选择

# 10 GBDT

参考如下

- 机器学习（四）--- 从gbdt到xgboost
- 机器学习常见算法个人总结（面试用）
- xgboost入门与实战（原理篇）

## 简介

> GBDT是一个基于迭代累加的决策树算法，它通过构造一组弱的学习器（树），并把多颗决策树的结果累加起来作为最终的预测输出。

## 算法介绍

**GBDT**是希望组合一组弱的学习器的线性组合，即有：

$$F^* = argminE_{x,y}[L(y, F(x))]$$
$$F(x; p_m, a_m) = \sum_{m=0}^{M} p_m h(x; a_m)$$

上述公式中$p_m$表示步长，我们可以在函数空间形式上使用梯度下降法求解，首先固定$x$，然后对$F(x)$求其最优解。下面先给出框架流程：

$$F^*(x) = argminE_y[L(y, F(x))|x]$$
$$F_0(x) = f_0(x)$$
$$for \quad m = 1 \cdots M:$$
$$g_m(x) = -\frac{\partial E_y[L(y, F(x))|x]}{\partial F(x)}|_{F(x)=F_{m-1}(x)} \qquad \text{descent direction}$$
$$\rho_m = argminE_y[L(y, F_{m-1}(x) + \rho g_m(x)|x] \qquad \text{step size}$$
$$f_m(x) = \rho_m g_m(x)$$
$$F_m(x) = F_{m-1}(x) + \rho_m g_m(x)$$
$$end \quad for$$
$$F^*(x) \approx F_M(x) = f_0(x) + \sum_{m=1}^{M} \rho_m g_m(x)$$



我们需要做的是估计$g_m(x)$,它是梯度方向；通过使用决策树实现来逼近$g_m(x)$，使得两者之间的距离尽可能的近，而距离的衡量方式有多种，包括均方误差和`LogLoss`误差。下面给出使用`LogLoss`损失函数的具体推导：

$$L(y, F) = log(1 + exp(-2yF)) \qquad y \in [-1, 1]$$

**Step1** 首先求解初始值$F_0$，令其偏导为0。（实现时是第1棵树需要拟合的残差）：

$$
\begin{aligned}
&F_0 = argmin \sum_{i=1}^{N} L(y_i, F) \\
&\frac{\partial \sum_{i=1}^{N} L(y_i, F)}{\partial F} = 0 \\
&\Rightarrow \sum_{i=1}^{N} \frac{exp\{-2y_iF\}(-2y_i)}{1+exp\{-2y_iF\}} = 0 \\
&\Rightarrow \sum_{i:y_i=1} \frac{-2exp\{-2F\}}{1+exp\{-2F\}} + \sum_{i:y_i=-1} \frac{2exp\{2F\}}{1+exp\{2F\}} = 0 \\
&\Rightarrow F_0(x) = \frac{1}{2} log \frac{1+\bar{y}}{1-\bar{y}}
\end{aligned}
$$



**Step 2** 估计$g_m(x)$，并用决策树对其进行拟合。$g_m(x)$是梯度，实现时是第$m$棵树需要拟合的残差：

$$
\begin{aligned}
g_m(x_i) &= -\frac{\partial L(y_i, F)}{\partial F}|_{F=F_{m-1}} \\
&= \frac{2y_i exp\{-2y_i F_{m-1}(x_i)\}}{1+exp\{-2y_i F_{m-1}(x_i)\}} \\
&= 2y_i / (1+exp\{2y_i F_{m-1}(x_i)\})
\end{aligned}
$$



**Step 3** 使用牛顿法求解下降方向步长。$r_jm$是拟合的步长，实现时是每棵树的预测值。（通常实现中这一步是被省略的，改为使用**Shrinkage**的策略通过参数设置步长，避免过拟合。

$$f(r) = \sum_{x_i \in R_{jm}} log(1 + exp(-2y_i(F_{m-1}(x_i) + r)))$$

$$f'(r) = \sum_{x_i \in R_{jm}} \frac{-2y_i}{1 + exp(2y_i(F_{m-1}(x_i) + r))}$$

$$f''(r) = \sum_{x_i \in R_{jm}} \frac{2y_i exp(2y_i(F_{m-1}(x_i) + r)}{[1 + exp(2y_i(F_{m-1}(x_i) + r))]^2}$$

$$\gamma_{jm} \approx \gamma_0 - f'(r_0)/f''(r_0) = \frac{\sum_{x_i \in R_{jm}} \tilde{y}_i}{\sum_{x_i \in R_{jm}} |\tilde{y}_i| (2 - |\tilde{y}_i|)}$$



**Step 4** 预测时只需要把每棵树的预测值乘以缩放因子然后相加即可得到最终的预测值：

$$p = predict(0) + \sum_{m=1}^{M} shrinkage * predict(d_m)$$

若需要预测值输出区间在$[0, 1]$，可作如下转换：

$$probability = \frac{1}{1 + e^{-2 * predict}}$$

**GBDT**中的树是回归树，不是分类树。

## RF与GBDT对比

（1）RF中树的棵树是并行生成的；GBDT中树是顺序生成的；两者中过多的树都会过拟合，但是**GBDT**更容易过拟合；

（2）RF中每棵树分裂的特征比较随机；GBDT中前面的树优先分裂对大部分样本区分的特征，后面的树分裂对小部分样本区分特征；

（3）RF中主要参数是树的棵数；GBDT中主要参数是树的深度，一般为**1**；

## Shrinkage

**Shrinkage**认为，每次走一小步逐步逼近的结果要比每次迈一大步逼近结果更加容易避免过拟合。

$$y(1 \sim i) = y(1 \sim i - 1) + step * y_i$$

# 优缺点

## 优点

1. 精度高
2. 能处理非线性数据

3. 能处理多特征类型
4. 适合低维稠密数据
5. 模型可解释性好
6. 不需要做特征的归一化，可以自动选择特征
7. 能适应多种损失函数，包括均方误差和`LogLoss`等

### 缺点

1. **boosting**是个串行的过程，所以并行麻烦，需要考虑上下树之间的联系
2. 计算复杂度大
3. 不使用高维稀疏特征

## 调参

1. 树的个数 100~10000
2. 叶子的深度 3~8
3. 学习速率 0.01~1
4. 叶子上最大节点树 20
5. 训练采样比例 0.5~1
6. 训练特征采样比例 $\sqrt{(n)}$

# xgboost

**xgboost**是**boosting Tree**的一个很牛的实现，它在最近Kaggle比赛中大放异彩。它 有以下几个优良的特性：

1. 显示的把树模型复杂度作为正则项加到优化目标中。
2. 公式推导中用到了二阶导数，用了二阶泰勒展开。
3. 实现了分裂点寻找近似算法。
4. 利用了特征的稀疏性。
5. 数据事先排序并且以block形式存储，有利于并行计算。
6. 基于分布式通信框架rabit，可以运行在MPI和yarn上。（最新已经不基于rabit了）
7. 实现做了面向体系结构的优化，针对cache和内存做了性能优化。

在项目实测中使用发现，Xgboost的训练速度要远远快于传统的GBDT实现，10倍量级。

### 特点

这部分内容参考了知乎上的一个问答—机器学习算法中**GBDT**和**XGBOOST**的区别有哪些？，答主是**wepon**大神

1.传统GBDT以CART作为基分类器，xgboost还支持线性分类器，这个时候xgboost相当于带L1和L2正则化项的逻辑斯蒂回归（分类问题）或者线性回归（回归问题）。 —可以通过**booster [default=gbtree]**设置参数:**gbtree: tree-based models/gblinear: linear models**

2.传统GBDT在优化时只用到一阶导数信息，xgboost则对代价函数进行了二阶泰勒展开，同时用到了一阶和二阶导数。顺便提一下，xgboost工具支持自定义代价函数，只要函数可一阶和二阶求导。 —对损失函数做了改进（泰勒展开，一阶信息**g**和二阶信息**h**）

3.xgboost在代价函数里加入了正则项，用于控制模型的复杂度。正则项里包含了树的叶子节点个数、每个叶子节点上输出的score的L2模的平方和。从Bias-variance tradeoff角度来讲，正则项降低了模型variance，使学习出来的模型更加简单，防止过拟合，这也是xgboost优于传统GBDT的一个特性
—正则化包括了两个部分，都是为了防止过拟合，剪枝是都有的，叶子结点输出**L2**平滑是新增的。

4.shrinkage and column subsampling —还是为了防止过拟合

> （1）shrinkage缩减类似于学习速率，在每一步tree boosting之后增加了一个参数n（权重），通过这种方式来减小每棵树的影响力，给后面的树提供空间去优化模型。
>
> （2）column subsampling列(特征)抽样，说是从随机森林那边学习来的，防止过拟合的效果比传统的行抽样还好（行抽样功能也有），并且有利于后面提到的并行化处理算法。

5.split finding algorithms(划分点查找算法)：
（1）exact greedy algorithm—贪心算法获取最优切分点
（2）approximate algorithm— 近似算法，提出了候选分割点概念，先通过直方图算法获得候选分割点的分布情况，然后根据候选分割点将连续的特征信息映射到不同的**buckets**中，并统计汇总信息。
（3）Weighted Quantile Sketch—分布式加权直方图算法
这里的算法（**2**）、（**3**）是为了解决数据无法一次载入内存或者在分布式情况下算法（**1**）效率低的问题，以下引用的还是**wepon**大神的总结：

> 可并行的近似直方图算法。树节点在进行分裂时，我们需要计算每个特征的每个分割点对应的增益，即用贪心法枚举所有可能的分割点。当数据无法一次载入内存或者在分布式情况下，贪心算法效率就会变得很低，所以xgboost还提出了一种可并行的近似直方图算法，用于高效地生成候选的分割点。

6.对缺失值的处理。对于特征的值有缺失的样本，xgboost可以自动学习出它的分裂方向。 **—**稀疏感知算法

7.**Built-in Cross-Validation**（内置交叉验证**)**

> XGBoost allows user to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run.
> This is unlike GBM where we have to run a grid-search and only a limited values can be tested.

8.**continue on Existing Model**（接着已有模型学习）

> User can start training an XGBoost model from its last iteration of previous run. This can be of significant advantage in certain specific applications.
> GBM implementation of sklearn also has this feature so they are even on this point.

9.**High Flexibility**（高灵活性）

> **XGBoost allow users to define custom optimization objectives and evaluation criteria.
> This adds a whole new dimension to the model and there is no limit to what we can do.**

10.并行化处理 **—**系统设计模块,块结构设计等

> xgboost工具支持并行。boosting不是一种串行的结构吗?怎么并行的？注意xgboost的并行不是tree粒度的并行，xgboost也是一次迭代完才能进行下一次迭代的（第t次迭代的代价函数里包含了前面t-1次迭代的预测值）。**xgboost**的并行是在特征粒度上的。我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），xgboost在训练之前，预先对数据进行了排序，然后保存为**block**结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个block结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行。

此外xgboost还设计了高速缓存压缩感知算法，这是系统设计模块的效率提升。
当梯度统计不适合于处理器高速缓存和高速缓存丢失时，会大大减慢切分点查找算法的速度。
（1）针对 exact greedy algorithm采用缓存感知预取算法
（2）针对 approximate algorithms选择合适的块大小

## 代码使用

下面给出简单使用**xgboost**这个框架的例子。

```
# 划分数据集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.01, random_state=1729)
print(X_train.shape, X_test.shape)

#模型参数设置
xlf = xgb.XGBRegressor(max_depth=10,
                       learning_rate=0.1,
                       n_estimators=10,
                       silent=True,
                       objective='reg:linear',
                       nthread=-1,
                       gamma=0,
                       min_child_weight=1,
                       max_delta_step=0,
                       subsample=0.85,
                       colsample_bytree=0.7,
                       colsample_bylevel=1,
                       reg_alpha=0,
                       reg_lambda=1,
                       scale_pos_weight=1,
                       seed=1440,
                       missing=None)

xlf.fit(X_train, y_train, eval_metric='rmse', verbose = True, eval_set = [(X_test,
y_test)],early_stopping_rounds=100)

# 计算 auc 分数、预测
preds = xlf.predict(X_test)
```

一个运用到实际例子的代码，来自[xgboost入门与实战（实战调参篇）]

```
import numpy as np
import pandas as pd
import xgboost as xgb
from sklearn.cross_validation import train_test_split

#from xgboost.sklearn import XGBClassifier
#from sklearn import cross_validation, metrics   #Additional scklearn functions
#from sklearn.grid_search import GridSearchCV   #Perforing grid search
#
#import matplotlib.pylab as plt
#from matplotlib.pylab import rcParams

#记录程序运行时间
import time
start_time = time.time()

#读入数据
train = pd.read_csv("Digit_Recognizer/train.csv")
tests = pd.read_csv("Digit_Recognizer/test.csv")

params={
'booster':'gbtree',
'objective': 'multi:softmax', #多分类的问题
'num_class':10, # 类别数，与 multisoftmax 并用
'gamma':0.1,  # 用于控制是否后剪枝的参数,越大越保守，一般0.1、0.2这样子。
'max_depth':12, # 构建树的深度，越大越容易过拟合
'lambda':2,  # 控制模型复杂度的权重值的L2正则化项参数，参数越大，模型越不容易过拟合。
'subsample':0.7, # 随机采样训练样本
'colsample_bytree':0.7, # 生成树时进行的列采样
'min_child_weight':3,
# 这个参数默认是 1，是每个叶子里面 h 的和至少是多少，对正负样本不均衡时的 0-1 分类而言
#，假设 h 在 0.01 附近，min_child_weight 为 1 意味着叶子节点中最少需要包含 100 个样本。
#这个参数非常影响结果，控制叶子节点中二阶导的和的最小值，该参数值越小，越容易 overfitting。
'silent':0 ,#设置成1则没有运行信息输出，最好是设置为0.
'eta': 0.007, # 如同学习率
'seed':1000,
'nthread':7,# cpu 线程数
#'eval_metric': 'auc'
}

plst = list(params.items())
num_rounds = 5000 # 迭代次数

train_xy,val = train_test_split(train, test_size = 0.3,random_state=1)
#random_state is of big influence for val-auc
y = train_xy[:, 0]
X = train_xy[:, 1:]
val_y = val[:, 0]
val_X = val[:, 1:]

xgb_val = xgb.DMatrix(val_X,label=val_y)
xgb_train = xgb.DMatrix(X, label=y)

xgb_test = xgb.DMatrix(tests)
```

```
watchlist = [(xgb_train, 'train'),(xgb_val, 'val')]

# training model
# early_stopping_rounds 当设置的迭代次数较大时，early_stopping_rounds 可在一定的迭代次数内准确率没有提升就停止训练
model = xgb.train(plst, xgb_train, num_rounds, watchlist,early_stopping_rounds=100)

model.save_model('./model/xgb.model') # 用于存储训练出的模型
print "best best_ntree_limit",model.best_ntree_limit

print "跑到这里了model.predict"
preds = model.predict(xgb_test,ntree_limit=model.best_ntree_limit)

np.savetxt('xgb_submission.csv',np.c_[range(1,len(tests)+1),preds],delimiter=',',header='ImageId,Label',comments='',fmt='%d')

#输出运行时长
cost_time = time.time()-start_time
print "xgboost success!",'\n',"cost time:",cost_time,"(s)"
```

所使用的数据集是Kaggle上的Classify handwritten digits using the famous MNIST data--手写数字识别数据集，即`Mnist`数据集。