中山大學
SUN YAT-SEN UNIVERSITY

《人工智能实验》

实验报告

（期中项目）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 计科 2 班
学 生 姓 名 ：	李新锐
学　　号	15323032
时　　间 ：	2018　年　10　月　21　日

一、数据预处理
数据清理
向量化
二、KNN
创新点
实验结果
三、朴素贝叶斯
算法原理
伪代码
实验创新
代码展示
实验结果
四、回归树
算法原理
数据划分
终止条件
伪代码
代码
算法改进
实验结果
五、随机森林
Bagging
伪代码
从Bagging到随机森林
伪代码
代码
实验结果
六、逻辑斯特回归
算法原理
伪代码
代码
实验结果

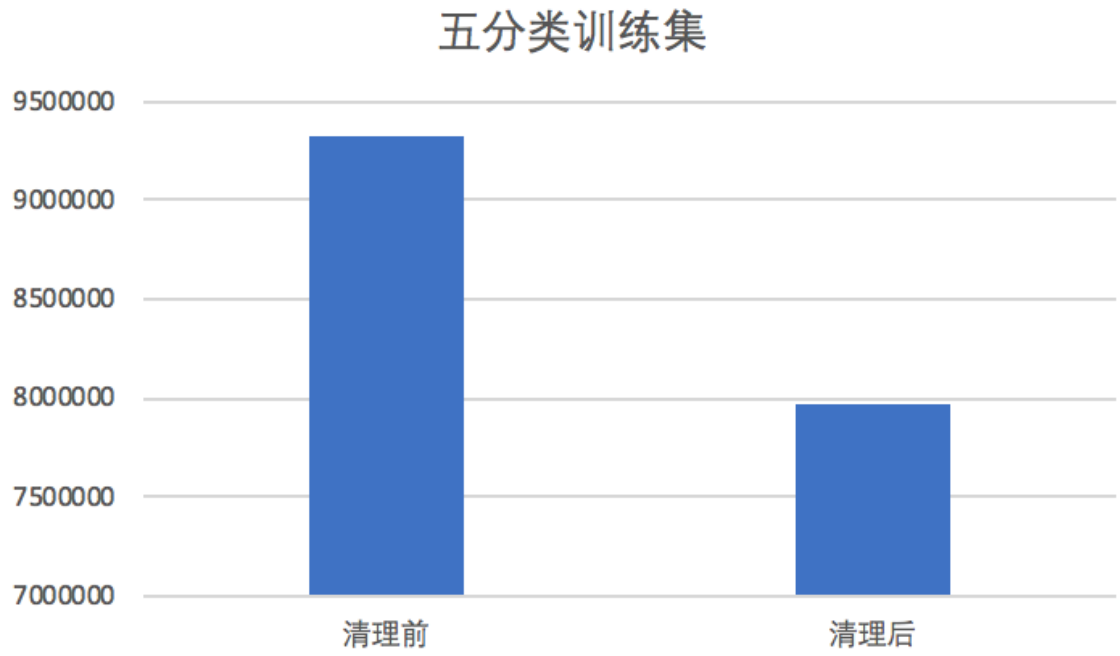
一、数据预处理

数据清理

对于二分类和五分类的训练样本，由于它们是原始的自然语言文本, 因此在向量化之前，我对它们进行了较为全面的数据清洗工具，包括了以下几个步骤：

1. 使用nltk.tokenize进行分词
2. 使用了isalpha函数去除非单词内容。

该步骤去除了文本中的数字、标点、特殊符号、HTML标签等非英文内容。以五分类数据集为例，清理前后训练集单词总量从93万下降到79万，降幅达15.



3. 进行拼写检查

   自然语言文本中有许多拼写错误，检测并更正这些错误能够减少文本中的噪声。

   我首选尝试的是是PyEnchant工具和aspell-en字典，然而人工检查发现它把很多人名识别为了错误。因此改用了autocorrect包，人工检查发现效果不错。

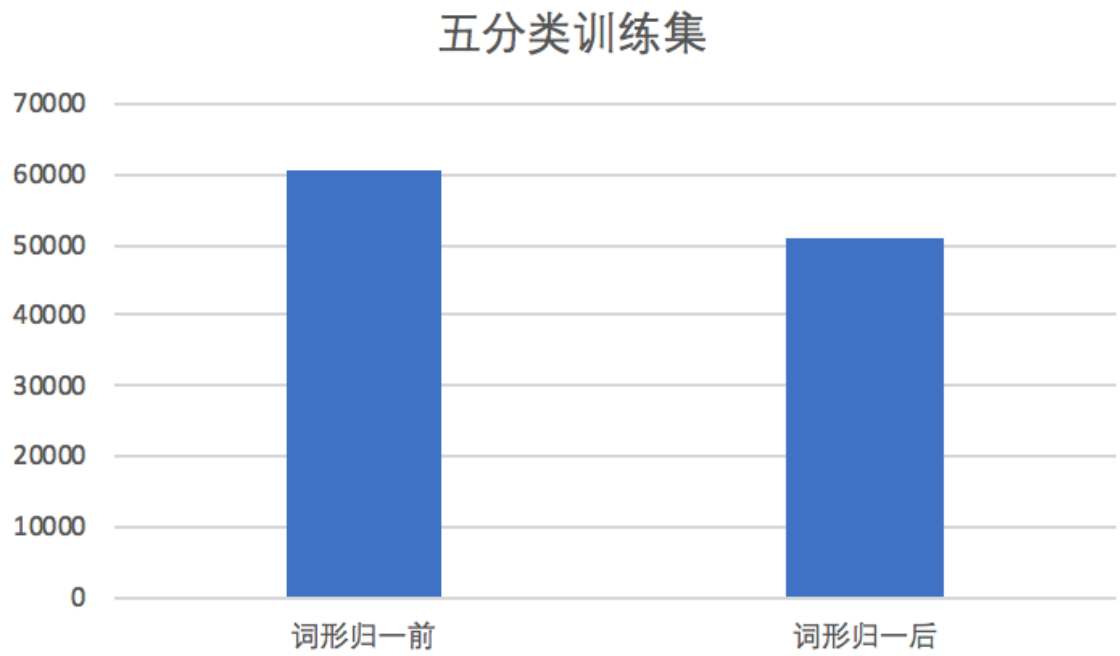
   以五分类数据集为例，拼写检查结果如下：

更正数目	**更正率**
187672	2.345%

4. 词形归一化（包括大写转小写）

   英语文本的一个特点是同一单词可能有着不同的形式。如take taken taking。在自然语言处理中这些词形信息往往不重要，通过词形归一化能够降低不同的单词数目，起到降维的效果。我使用的归一化工具是nltk.stem.WordNetLemmatizer。

   在五分类训练集上词形归一化前后文本中不同的单词数目从60760下降到了51169。相当于降维了近1万维。



向量化

本次实验中我主要使用了Onehot矩阵和doc2vec矩阵。其中前者是在实验一中实现的，后者调用了genism库。该方法和实验课上所讲的word2vec类似，区别在于训练集中每个句子被分配一个id，在训练模型的输入层中和句子中所有单词放在一起训练，相当于在训练的过程中用到了整个句子的信息。训练的结果被当做整个句子的向量表示。

最终得到的向量表示如下：

数据集	维度
二分类-onehot	62751
二分类-doc2vec	50
五分类-onehot	51170
五分类-doc2vec	50

二、KNN

创新点

本次实验中，我首先尝试了直接使用lab1中实现的KNN算法，由于样本个数过大，预测速度非常缓慢。因此作出一处改进：将计算L2范数的过程改为矩阵运算。原理如下：

设 X 为 N x M 的训练集，T为PxM的测试集。

目标是得到PxN的距离矩阵D。D中元素$D_{i,j}$是测试集向量T_i和训练集中向量X_i的距离。则有

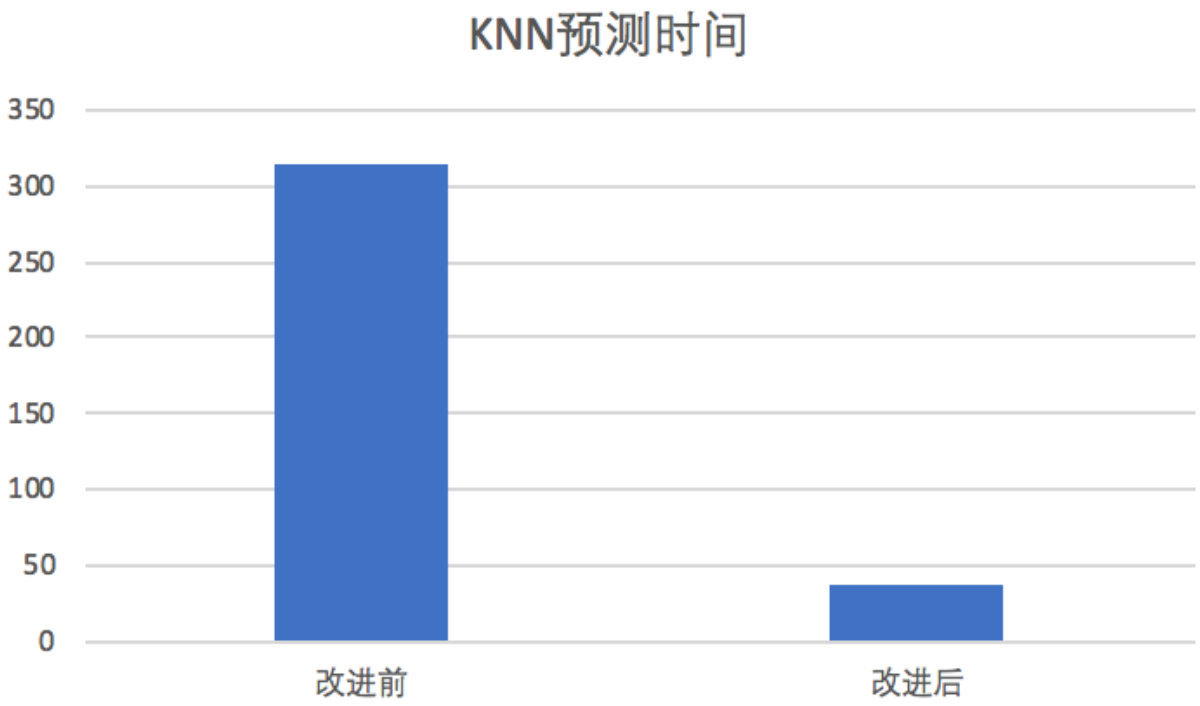
$$D_{i,j} = \sqrt{\sum_{k}^{m}(X_{i,k} - T_{j,k})^2}$$

代码实现：

```
#trainSet, testSet, dists分别对应原理说明中的X,T,D矩阵
#trainSum: N x 1
trainSum = np.sum(np.square(trainSet), axis=1)
#testSum: P x 1
testSum = np.sum(np.square(testSet), axis=1)
#t0:P x N
t0 = np.dot(testVec, trainSet[0].T)
#dist: P x N
dists = np.sqrt(-2 * t0 + testSum.reshape(-1, 1)+ trainSum)
```

实验结果

我使用了doc2vec矩阵，在二分类训练集中，选取后4000条作为验证集。采用上述的改进方式前后，预测时间从5分降低到34秒左右。



但缺点是，上述改进限制了只能使用二范数。因此可调参数只有K值。在K值从1到13的过程中，二分类 分类准确度变化如下：

可见，准确度均为达到50%。可能原因是doc2vec的降维并不是一个保范数的过程，50维的doc2vec样本空间中的样本直接的空间相对远近和原始空间中各样本之间的相对远近是不一致的。

因此，我尝试改用onehot矩阵，但二分类的onehot矩阵有62751维，用KNN无法在合理的时间内处理。因此放弃了并寻求更好的方法。

三、朴素贝叶斯

算法原理

本次实验中我尝试的第二种算法是朴素贝叶斯算法。

朴素贝叶斯是一种分类算法，基于的是概率论中贝叶斯公式。

假设我们有N个样本，每个样本有m个特征。样本分为K各类别$C_1, C_2, \ldots, C_K$。分类问题要求的是在输入为X的条件下输出为Y_k的概率。由条件概率公式得：

$$P(Y_k|X) = \frac{P(Y_k X)}{P(X)}$$

而$P(Y_k X)$可以写作$P(X|Y_k)P(Y_k)$，同时由全概率公式知

$$P(X) = \sum_k P(Y_k)P(X|Y_k)$$

因此我们就得到了贝叶斯公式：

$$P(Y_k|X) = \frac{P(X|Y_k)P(Y_k)}{\sum_k P(X|Y_k)P(Y_k)}$$

而如何利用该公式进行分类呢，方法如下：

1. 计算样本属于各个类别的先验分布

$$P(Y=C_K)$$

2. 计算条件概率$P(X|Y=C_K)$。假设X的各个特征之间是独立的。可得

$$P(X|Y=C_K)=P(X_1=x_1|Y=C_k)P(X_2=x_2|Y=C_k)\\ *\ldots*P(X_m=x_m|Y=C_k)$$

3. 由于贝叶斯公式的分母$P(X)$是一样的，因此预测分类结果C^*是使得分母最大化的Y的取值。因此贝叶斯分类的问题就变为了求解：

$$argmax_{C_k}P(X|Y=C_K)*P(Y=C_K)\\ =argmax_{C_k}P(X_1=x_1|Y=C_k)P(X_2=x_2|Y=C_k)\\ *\ldots*P(X_m=x_m|Y=C_n)*P(Y=C_K)$$

4. 最后要解决的问题如何计算有关X的各个特征m的条件概率$P(X_m|Y=C_k)$。对于特征取值离散的oneHot矩阵，可以直接统计频率构造伯努利分布。然而使用doc2vec进行向量化后，各个维度X_s都是连续值。因此处理方法是：假设各个特征都服从正态分布，通过样本计算出均值和方差，即可拟合出这个分布。

伪代码

oneHot版本：

```
# 记训练集输入为D，标签集为Y，要预测的输入为X
# 计算K个先验概率
for k in 1...K:
    P_Y[k] = count(Y == k) / count(D)
# 按照类别划分输入集
for k in 1...K:
    D_parted[k] = filter(Y == k, D)
# 统计每一维度特征在K种类别上的伯努利分布
PXY = [[1-sum(attr)/N, sum(attr)/N] for attr in zip(*D_parted[k])]

# 计算输入X为K种类别的概率
P = P_Y
for k in 1...K:
    for i in 1...m:
        P[k] *= PXY[k][i][X[i]] #P(X_i = X_i | y_k)
# 取概率最大的一种作为输出
return max_argument(P)
```

doc2Vec版本与oneHot版本的区别在于

```
# 第八行改为
gauss = ([Guass(mean(attr), var(attr)) for attr in zip(*D_parted[k])])
# 统计每一维度特征在K种类别上的均值和方差，并依此构造高斯分布

# 第15行改为
P[k] *= gauss[k][i](X[i])
# 计算高斯分布在X[i]点的概率
```

实验创新

由于onehot矩阵的维度非常高，在代码第15行连续计算乘法会导致最终得到的概率被舍为0.因此改为取对数求加法。

```
P[k] += log(PXY[k][i][X[i]])
```

然而$PXY[k][i][X[i]]$的值可能为0，会导致算出nan的情况。因此又要对PXY的计算进行改进，将第9行改为

```
1-(1+ sum(attr))/(N+2), (1+sum(attr))/(N+2)
```

来避免这种情况。

代码展示

```
## D_parted[k] 对应分类为k的样本集
D_parted = list(range(K))
for k in range(2):
    ## 通过filter函数筛选
    D_parted[k] = np.array(list(filter(lambda v: v[-1] == k, D)))
    ## 计算先验概率
    P_Y[k] = len(D_parted[k]) / len(D)

## 定义高斯分布类
class Guass:
    # 均值
    mean  = 0
    #标准差
    stdev = 0
    def __init__(self, stats):
        self.mean, self.stdev = stats
    # 计算x处的概率
    def P(self, x):
        t1 = np.exp(-(np.power(x-self.mean,2)/(2*np.power(self.stdev,2))))
        return (1 / (np.sqrt(2*np.pi) * self.stdev)) * t1


guass = list(range(K))
# 统计每一维度特征在K种类别上的均值和方差，并依此构造高斯分布
for k in range(K):
    guass[k] = [Guass((np.mean(attr), np.std(attr))) for attr in zip(*D_parted[k])]
    # 删除最后一列(Y值)构造的高斯分布
    del guass[k][-1]

# 统计每一维度特征在K种类别上的伯努利分布，和上面的代码一样，为了使用tqdm库显示运行进度条才拆开了写，所有看起来长一点
onehotPXY = list(range(K))
for k in range(K):
    l = list(range(len(X[0]) + 1))
```

```
    z = zip(*D_parted[k])
    for i, attr in enumerate(tqdm_notebook(z)):
        l[i] = (1-(sum(attr)+1)/(len(D_parted[k])+2), (sum(attr)+1)/(len(D_parted[k])))
    del l[-1]
    onehotPXY[k] = l

# 预测部分
def predict(X):
    # 使用deepcopy防止修改P_Y
    _PY = deepcopy(P_Y)
    # 计算输入X为K种类别的概率
    for k in range(K):
        for i in range(M):
            ret = np.log(guass[k][i].P(X[i]))
            _PY[k] += ret
    #取概率最大的一种作为输出
    return np.argmax(_PY)
```

实验结果

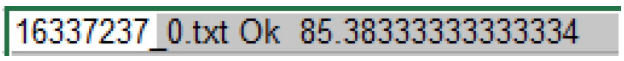
采用K交叉验证

使用doc2vec仅有31%的正确率。推测原因是：输入向量各维度是正态分布的假设和doc2vec向量的实际不同。

使用onehot时正确率非常好，5-交叉和6-交叉验证的正确率均为90.88%。

K值	正确率
5	90.8875%
6	90.8875%

提交rank后，在测试集上的准确率为85.38，比在验证集上低5%左右。这说明验证集上的数据与测试集分布式有一定差别的。由于



四、回归树

由于朴素贝叶斯算法缺少调参空间，在得到测试结果后，我接着尝试了改进实验二中实现的决策树。当时我只实现了处理离散数据的分类树，但doc2vec矩阵各个维度上都是连续值，因此我实现了回归树算法。算法原理如下：

算法原理

数据划分

将测试数据依据某个特征a，以s为切分点进行切分。求解

$$argmin_{a,s}[\sum_{x_i<s}(avg_1-y_i)^2+\sum_{x_i\geq s}(avg_2-y_i)^2]$$

其中avg_1是s左边数据的平均值

avg_2是s及右边数据的平均值

每个区域的输出即是平均值

终止条件

1. 误差函数的值差距小于阈值

$$\sum_{i\in D}(avg-y_i)^2<tolerance$$

2. 选择最好的分割方法,不纯度减少量仍低于阈值

$$\sum_{x_i<s}(avg_1-y_i)^2+\sum_{x_i\geq s}(avg_2-y_i)^2-\sum_{i\in D}(avg-y_i)^2<tolerance$$

3. 数据量小于阈值

$$|D|<toleracnce$$

伪代码

```
# 依据训练样本集D和特征集A训练回归树
# errFunc可以采用方差
train(D, errFunc):
    # 训练集D的X，Y分开
    (X, Y) = D
    # 记avg为当前节点数据集的结果的平均数
    avg = sum(Y) / |D|
    '''
    结束条件有3种，分别是：
        1.误差函数的值差距小于阈值
        2.选择最好的分割方法,不纯度减少量仍低于阈值
        3.数据量小于阈值
    '''
    if errFunc(D) < tolreance1 or |D| < tolreance2
        # 设置当前节点为叶节点，结果为mode
        node.isLeaf = True
        node.Y = avg
        return node

    bestA = -1, bestErr = errFunc(D)
    # 遍历一种特征
    for a in A:
```

```
        # 对每一种该特征的取值
        for s in X[a]:
            # 划分数据集为两份
            D_l, D_r = split_data(s)
            # 计算新误差
            newErr = errFunc(D_l) + errFunc(D_r)
            #记录最佳划分
            if deltaErr < bestErr:
                bestErr = deltaErr
    # 选择最好的分割方法,不纯度减少量仍低于阈值
    if bestA == -1:
        node.isLeaf = True
        node.Y = avg
        return node
    #递归训练子节点
    node.child.append(train(D_l))
    node.child.append(train(D_r))
```

代码

```
// 实际代码中考虑了深度限制，因此多了int depth（当前深度），int level（最大深度）两个参数
void RegressionTree::train_worker(RegressionTree::Node *node, const ErrFunc_t &errFunc,
int depth, int level)
{
    //训练集D的X，Y分开
    auto* pA = &node->A;
    auto* pD = &node->D;
    auto D_Y = pD->row(-1);
    // 记avg为当前节点数据集的结果的平均数
    auto avg = D_Y.sum() / D_Y.size();
    //当前误差
    auto err = errFunc(D_Y, avg);
    //cout << "In level " << depth << endl;
    /*
    结束条件有3种，分别是：
        1.误差函数的值差距小于阈值
        2.选择最好的分割方法,不纯度减少量仍低于阈值
        3.数据量小于阈值
    */
    if(err < RegressionArgs::err_tolerance || depth > level)
    {
        node->isLeaf = true;
        node->Y = avg;
        return;
    }

    int bestA = -1;
    double bestS = INFINITY;
    double bestErr = err;
```

```
    // 遍历一种特征
    for(const auto& a : range(0, pA->size()))
    {
        auto vals = pD->row(a);
        std::set<double> vals_set;
        auto n = vals.size();
        //实际代码中，只抽样该特征的部分取值
        for(Eigen::Index i = 0; i < 10; ++i)
        {
            vals_set.insert(vals[RandLib::uniform_rand(0, n - 1)]);
        }
        for(auto s: vals_set)
        {
            //划分数据集
            auto[D_l, D_r] = split_data(node->D, a, s);
            auto Y_l = D_l.row(-1);
            auto Y_r = D_r.row(-1);
            auto avg_l = Y_l.sum() / Y_l.size();
            auto avg_r = Y_r.sum() / Y_r.size();
            auto err_l = errFunc(Y_l, avg_l);
            auto err_r = errFunc(Y_r, avg_r);
            //计算误差变化
            auto new_err = err_l + err_r;
            if(new_err < bestErr)
            {
                bestA = a;
                bestS = s;
                bestErr = new_err;
            }
        }
    }
    //cout << "Best cut: new_err = " << bestErr << " , A = " << bestA << " , S = " <<
bestS << endl;
    //选择最好的分割方法,不纯度减少量仍低于阈值，则结束递归
    if(bestA == -1)
    {
        node->isLeaf = true;
        node->Y = avg;
        return;
    }
    auto[D_l, D_r] = split_data(node->D, bestA, bestS);
    //判断左右子节点是否有数据集为空
    if(D_l.empty())
    {
        node->isLeaf = true;
        node->Y = avg;
        return;
    }
    if(D_r.empty())
```

```
    {
        node->isLeaf = true;
        node->Y = avg;
        return;
    }
    //否则递归训练子节点
    node->C = bestA;
    node->S = bestS;
    node->ch_l = new Node(D_l, *pA);
    node->ch_r = new Node(D_r, *pA);
    train_worker(node->ch_l, errFunc, depth+1, level);
    train_worker(node->ch_r, errFunc, depth+1, level);
}
```

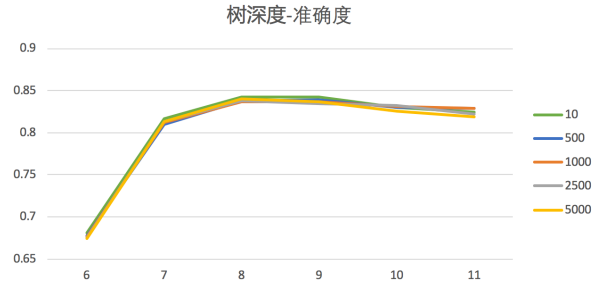
算法改进

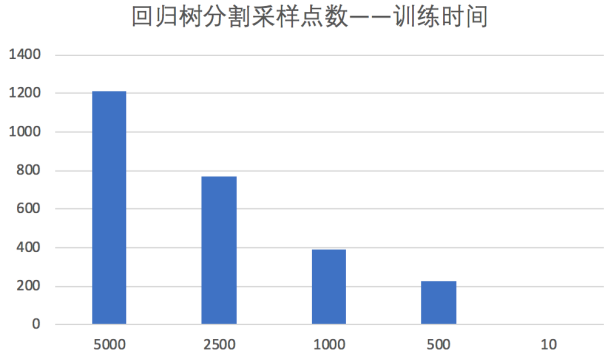
如上述代码所示，在实际实现回归树时，并不是遍历某一特征所有取值，而只是随机抽取部分点，从而降低训练决策树的时间。这种分割可能无法找到最优分割点，如下实验结果所示：即便仅抽取10个分割点，在树达到一定深度时，分类效果不比抽取更多分割点差。

实验结果

- 测试方法：K=5的K交叉验证
- 测试对象：二分类问题（测试集大小：19200，验证集大小：4800）
- 测试输入：50维的doc2vec矩阵
- 图像中不同直线含义：在选取特征分割点时分别采样10，500，1000，2500，5000个点
- 横轴含义：树的最大深度
- 纵轴含义：准确度

可见，在树深度为8至9时有着最佳的分类准确度。其中使用5000个分割采样点的直线在深度为8时有准确度最大值84.0362，然而其他方法也与他相差无几。但带来的好处是近似线性的训练速度提升。





（注：10对应的训练时间为4，所有时间单位为秒）

五、随机森林

在实现了回归树算法后，我不断调整树的最大深度和采样点树，然而似乎训练效果不能继续提升了。因此我尝试采用了随机森林算法。

算法原理如下：

Bagging

随机森林算法的基础是Bagging。该算法对大小为N的原数据集进行N次放回的采样。由于有放回这一特点，导致有些点会被采样多次，有些点无法采样到，因此能够减少算法的过拟合。

伪代码

```
rets = 0
for i in range(M):
    # D是样本集
    sub_D = sample_D(D)
    # A是特征集
    tree = CART.train(sub_D, A)
    ret = tree.vaild(vaild_set)
    rets += ret
rets /= M
```

从Bagging到随机森林

由于回归树是一个贪婪算法，即便使用Bagging，得到的树在结构上仍然相似，结果高度相关。为了减少模型间的相关度，随机森林算法做出了进一步改进，在特征集上也进行抽样。

伪代码

```
rets = 0
for i in range(M):
    # D是样本集
    sub_D = sample_D(D)
    # A是特征集
    sub_A = sample_A(A)
    tree = CART.train(sub_D, sub_A)
    ret = tree.vaild(vaild_set)
    rets += ret
rets /= M
```

代码

```
// 对样本集trainSet采样
matrix_view<double> BaggingRegressTree::sample_D(const matrix_view<double>& trainSet)
```

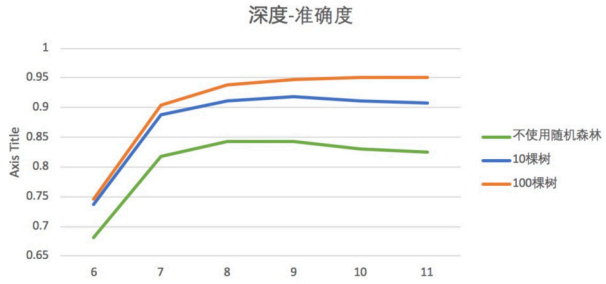
```cpp
{
    //v存储被采样的样本的下标
    std::vector<Eigen::Index> v;
    //使用集合类型自动去除重复
    std::set<Eigen::Index> s;
    auto n = trainSet.cols();
    //随机采样n次。RandLib::uniform_rand是我封装的均匀分布函数
    for(Eigen::Index i = 0; i < n; ++i)
        s.insert(RandLib::uniform_rand(0, n - 1));
    //返回采样到的样本
    for(auto i : s)
        v.push_back(i);
    return matrix_view<double>(trainSet, v);
}
// 从大小为n的特征空间中抽取k个特征
Vec<Eigen::Index> BaggingRegressTree::sample_A(size_t n, size_t k)
{
    Vec<Eigen::Index> ret;
    //若k大于等于n直接返回
    if(k >= n)
        return range(Eigen::Index(0), Eigen::Index(n));
    std::set<Eigen::Index> sel;
    //循环抽取直到抽取到k个不同的特征
    while(sel.size() < k)
    {
        int s = RandLib::uniform_rand(0, n - 1);
        sel.insert(s);
    }
    //返回被抽取的特征
    for(auto i : sel)
        ret.push_back(i);
    std::sort(ret.begin(), ret.end());
    return ret;
}
//训练随机森林，M代表森林中树的数量，k代表抽取的特征数
void BaggingRegressTree::train(const ErrFunc_t& errFunc, int M, int k, int maxLevel)
{
    //建立M棵树
    for(int i = 0; i < M; ++i)
    {
        //样本集采样
        matrix_view<double> sub_D = sample_D(trainSet);
        //特征集采样
        Vec<Eigen::Index> sub_A_idx = sample_A(featureCount, k);
        //储存到随机森林类中
        sub_A_idx.push_back(-1);
        forests_A.push_back(sub_A_idx);
        sub_D.select_row(sub_A_idx);
        //建树
```

```
52          RegressionTree* tree = new RegressionTree(sub_D);
53          forests.push_back(tree);
54          //训练
55          tree->train(errFunc, maxLevel);
56      }
57  }
```

实验结果

- 测试方法：K=5的K交叉验证
- 测试对象：二分类问题（测试集大小：19200，验证集大小：4800）
- 测试输入：50维的doc2vec矩阵
- 图像中不同直线含义：不使用随机森林，随机森林建10棵树，建100棵树
- 横轴含义：树的最大深度
- 纵轴含义：准确度

如图可视：建100棵树的随机森林算法取得了出乎意料的好结果，在$K \geq 9$验证集分类准确度稳定达到95%！



六、逻辑斯特回归

最后，我尝试并实现了逻辑斯特回归算法。

算法原理

逻辑斯特回归是一种分类算法，它的原理是在样本空间中划分线性边界$a = W^T X$，再通过激活函数

$$h(x) = \frac{1}{1 + e^{-x}}$$

将边界两端的点映射到$0, 1$两种分类上。

从上面的描述可以看到，逻辑斯特回归模型的训练实际上就是去找到一个最佳的W矩阵。根据最大似然估计，可以定义模型的损失函数为

$$C(W) = -\sum_{i=1}^{n}(y_i log\, h(W^T x_i) + (1 - y_i) log\, (1 - W^T x_i))$$

通过梯度下降法可以对W的每个维度j分别进行更新

$$W_{new}^j = W^j - \mu\frac{\partial C(W)}{\partial W^j}$$
$$= W^j - \mu\sum_{i=1}^{n}[(h(W^TX_i) - y_i)X_i^j]$$
$$= W^j - \mu X^T(h(XW) - Y)$$

公式中X_i和y_i是大小为n的样本集中的样本

更新的结束条件可以设置为：

1. 限制最大迭代次数 max_epoch
2. W的变化小于阈值
3. $C(W)$的变化小于阈值

伪代码

```
# W初始化为M+1维的全1向量,
W = ones(M+1)
C = Cost(W)
epoch = 0
while True:
    # err[i] 是梯度下降法公式中h(W^T * X_i) - y_i这一项
    err = h(X * W) - Y
    # 更新W
    W = W - u * X.T * err
    #判断结束条件
    epoch += 1
    if epoch > max_epoch or delta(W) < min_change or delta(C) < min_change:
        break
```

代码

```
# sigmod函数
def h(x):
    return 1 / (1 + np.exp(-x))
# 最大似然cost函数
def cost(W):
    ret = 0
    # 简单的公式翻译
    for i in range(N):
        t0 = np.dot(W.T, X[i])
        t1 = Y[i] * np.log(h(t0))
        t2 = (1-Y[i]) * np.log(h(1-t0))
        ret -= (t1 + t2)
    return ret

# 最大迭代次数
max_epoch = 10000
# 记录最新的W
LastW = [0]
```

```
# 记录训练过程中的loss变化
loss = []
def train(W):
    epoch = 0
    while True:
        # X: N x M
        # W: M * 1
        # err[i] 是梯度下降法公式中h(W^T * X_i) - y_i这一项
        err = h(X @ W) - Y
        # 更新W
        #X.T: M x N
        #err: N x 1
        #X.T @ err : M x 1
        #学习率
        u = 0.00001
        deltaW = - u * (X.T @ err)
        # 更新并记录新的W
        W = W + deltaW
        LastW[0] = W
        # 记录新cost
        newCost = cost(W)
        loss.append(newCost)
        print(epoch, newCost)
        #判断结束条件
        epoch += 1
        if epoch >= max_epoch:
            break
```

实验结果

- 测试方法：K交叉验证
- 测试对象：二分类问题（测试集大小：19200，验证集大小：4800）
- 测试输入：50维的doc2vec矩阵

在经过多次调整学习率之后，我的逻辑斯特回归模型仍未能收敛，而是陷入了loss约为8520的局部极小点。

K值	正确率
5	78.208%
6	78.208%