硕士第8周作业

1 朴素贝叶斯：

利用朴素贝叶斯方法，计算收入为中等，年龄在20-40之间的本科生是否应该贷款（需要计算过程+代码）

编号	收入	学历	年龄	贷款
1	高	研究生	40-60	是
2	高	本科	>60	否
3	高	专科	20-40	是
4	中	研究生	40-60	是
5	中	本科	40-60	是
6	中	专科	20-40	是
7	中	研究生	>60	是
8	低	本科	>60	否
9	低	本科	20-40	是
10	低	专科	40-60	否
11	低	专科	>60	否

计算收入中等，20-40本科生是否应该贷款
（计算过程+代码）

记：
收入：高，中，低 > 2，1，0
学历：研究生，本科，专科 > 2，1，0
年龄：>60，40-60，20-40 > 2，1，0
有：

```
X = np.array([
    [2,2,1],
    [2,1,2],
    [2,0,0],
    [1,2,1],
    [1,1,1],
    [1,0,0],
    [1,2,2],
    [0,1,2],
    [0,1,0],
    [0,0,1],
    [0,0,2]
```

```
])
y = np.array([1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0])
```

所问问题收入中等，20-40本科生为[1,1,0]
因为收入中等未贷款的样本数为0，以及年龄20-40未贷款的样本数也为0，因此需要使用拉普拉斯平滑

手动计算

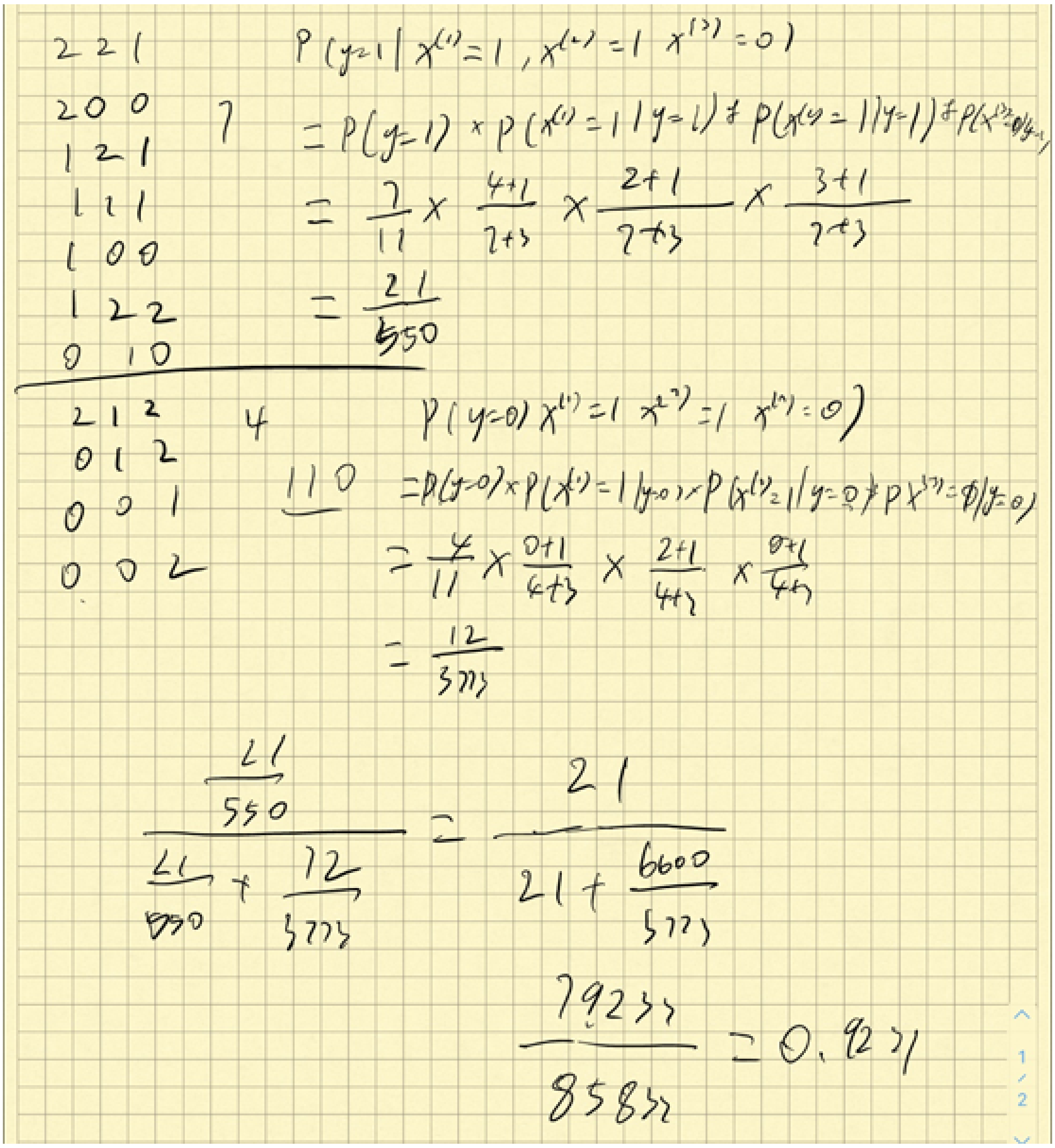
计算过程

$$
\begin{aligned}
& p(y=1|x^{(1)}=1, x^{(2)}=1, x^{(3)}=0) \\
& = p(y=1) \times p(x^{(1)}=1|y=1) \times p(x^{(2)}=1|y=1) \times p(x^{(3)}=0|y=1) \\
& = \frac{7}{11} \times \frac{4+1}{7+3} \times \frac{2+1}{7+3} \times \frac{3+1}{7+3} \\
& = \frac{21}{550}
\end{aligned}
$$

$$
\begin{aligned}
& p(y=0|x^{(1)}=1, x^{(2)}=1, x^{(3)}=0) \\
& = p(y=0) \times p(x^{(1)}=1|y=0) \times p(x^{(2)}=1|y=0) \times p(x^{(3)}=0|y=0) \\
& = \frac{4}{11} \times \frac{0+1}{4+3} \times \frac{2+1}{4+3} \times \frac{0+1}{4+3} \\
& = \frac{12}{3773}
\end{aligned}
$$

$$
P(y'=1) = \frac{\frac{21}{550}}{\frac{21}{550} + \frac{12}{3773}} = \frac{79233}{85833} = 0.9231
$$

$$
P(y'=0) = \frac{\frac{12}{3773}}{\frac{21}{550} + \frac{12}{3773}} = \frac{6600}{85833} = 0.0769
$$



结果

即贷款的概率为0.9231，不贷款的概率为0.07689，因此收入中等，20-40本科生**应贷款**

代码

手动计算代码

```
import numpy as np
X = np.array([
    [2,2,1],
    [2,1,2],
    [2,0,0],
    [1,2,1],
    [1,1,1],
    [1,0,0],
    [1,2,2],
    [0,1,2],
```

```
    [0,1,0],
    [0,0,1],
    [0,0,2]
])

y = np.array([1,0,1,1,1,1,1,0,1,0,0])

# 朴素贝叶斯类，面向离散特征
class NaiveBayes:
    def __init__(self):
        self.prior = {}
        self.prior_num = {}
        self.posterior = {}

    # 训练模型
    def fit(self, X, y):

        # 计算先验概率  数据结构形式：{0: 0.5, 1: 0.5}
        labels = list(set(y))
        self.prior = {label: 0 for label in labels}
        self.prior_num = {label: 0 for label in labels}
        for value in y:
            self.prior[value] +=1.0/ (len(y)) #累计计算先验概率
            self.prior_num[value] += 1 #累计不同分类的样本个数，备用

        self.prior_num[0] += 3
        self.prior_num[1] += 3
        # print(self.prior_num)

        # 计算后验概率  数据结构形式：{0: [{0: 0.75, 1: 0.25}, {0: 0.5, 1:
0.5}......}
        self.posterior = {label: [] for label in labels}
        for label in labels:
            for _ in range(X.shape[-1]):
                self.posterior[label].append({}) # 为每个类别，初始化一个空字典
        for item, label in zip(X, y):
            prior_num_y = self.prior_num[label]
            for i, val in enumerate(item):
                if val in self.posterior[label][i]:
                    self.posterior[label][i][val] += 1.0/prior_num_y  #已存在该特征
值，则进行累计计算
                else:
                    self.posterior[label][i][val] = 1.0/prior_num_y  #不存在该特征
```

```
值，则初始化

        print("先验概率:",self.prior)
        print("后验概率:",self.posterior)

    # 通过已知的概率，对样本做预测
    def predict_single(self, X_test):
        results = {}
        # 通过朴素贝叶斯公式来计算
        for label,prior_val in self.prior.items():
            results[label] = prior_val
            for i, post_val in enumerate(X_test):
                if(post_val in self.posterior[label][i]):
                    results[label] *= (self.posterior[label][i][post_val] + 1 /
self.prior_num[label])
                else:
                    results[label] *= 1 / self.prior_num[label]
        denominator = np.sum(list(results.values()))
        # 返回不同分类的概率
        for label in results.keys():
            results[label] = results[label]/denominator
        return results
nb = NaiveBayes()
nb.fit(X,y)
test_sample = [1,1,0]
print("测试结果:", nb.predict_single(test_sample))
```

运行结果

```
先验概率: {0: 0.36363636363636365, 1: 0.6363636363636365}
后验概率: {0: [{2: 0.14285714285714285, 0: 0.42857142857142855}, {1: 0.2857142857142857, 0: 0.2857142857142857}, {2: 0.42857142857142855, 1: 0.14285714285714285}], 1: [{2: 0.2, 1: 0.4, 0: 0.1}, {2: 0.30000000000000004, 0: 0.2, 1: 0.2}, {1: 0.30000000000000004, 0: 0.30000000000000004, 2: 0.1}]}
测试结果: {0: 0.07689350249903878, 1: 0.9231064975009613}
```

使用sklearn代码

```
from sklearn.naive_bayes import CategoricalNB
import numpy as np

X = np.array([
    [2,2,1],
    [2,1,2],
    [2,0,0],
    [1,2,1],
    [1,1,1],
    [1,0,0],
```

```
    [1,2,2],
    [0,1,2],
    [0,1,0],
    [0,0,1],
    [0,0,2]
])

y = np.array([1,0,1,1,1,1,1,0,1,0,0])


mnb = CategoricalNB(alpha = 1)
mnb.fit(X,y)

test_sample = np.array([[1,1,0]])
print("测试结果:", mnb.predict_proba(test_sample))
print("预测结果:", mnb.predict(test_sample))
```

运行结果

```
测试结果: [[0.0768935 0.9231065]]
预测结果: [1]
```

2 集成学习:

将数据集换成load_digits

```
from sklearn.datasets import load_digits
X, y = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

在bagging(随机森林)，和boosting(AdaBoost)中用不同数目的学习器（例如10,50,100）进行学习训练，打印结果的准确性

bagging(随机森林)

代码

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
X, y = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
nb_trees = [10,50,100]
for nb_tree in nb_trees:
    rf_clf = RandomForestClassifier(n_estimators=nb_tree, n_jobs=-1)
```

```
    rf_clf.fit(X_train, y_train)
    print("学习器为", nb_tree, "的准确性为：", rf_clf.score(X_test, y_test))
```

结果

```
学习器为 10 的准确性为： 0.96
学习器为 50 的准确性为： 0.98
学习器为 100 的准确性为： 0.9777777777777777
```

boosting(AdaBoost)

代码

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
X, y = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
nb_trees = [10,50,100]
for nb_tree in nb_trees:
    clf = AdaBoostClassifier(n_estimators=nb_tree, learning_rate=0.5)
    clf.fit(X_train, y_train)
    print("学习器为", nb_tree, "的准确性为：", clf.score(X_test, y_test))
```

结果

```
学习器为 10 的准确性为： 0.5666666666666667
学习器为 50 的准确性为： 0.6066666666666667
学习器为 100 的准确性为： 0.6711111111111111
```

3 线性模型

实现后向特征选择法

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import copy
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
X, y = load_boston(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
reg = LinearRegression()
reg.fit(X_train, y_train)
y_pred = reg.predict(X_train)
mse_dic = mean_squared_error(y_train, y_pred)

num_feature = X_train.shape[1]
```

```
feature_li=list(range(0,num_feature))
print(f"所有特征{feature_li}都进行训练模型的MSE误差为：{mse_dic}")
last_mse = mse_dic
for i in range(len(feature_li)):#一共有feature_li_num个数量的变量等待剔除
    remove_mse_dic = {}
    for feature in feature_li:
        if len(feature_li) < 2: break
        feature_li_wait_remove = copy.deepcopy(feature_li)
        # 去掉features组合中一个特征
        feature_li_wait_remove.remove(feature)
        X_in = X_train[:, feature_li_wait_remove]
        reg.fit(X_in, y_train)
        y_pred = reg.predict(X_in)
        remove_mse_dic[feature] = mean_squared_error(y_train, y_pred)
        # print(f"用特征{feature_li_wait_remove}进行训练模型的MSE误差为：
{remove_mse_dic[featu]}")
        if remove_mse_dic :
            max_fea = min(remove_mse_dic, key=remove_mse_dic.get)

        if remove_mse_dic[max_fea] < last_mse:
            last_mse = remove_mse_dic[max_fea]
            feature_li.remove(max_fea)
            print("第" + str(i+1 ) + "轮剔除特征",max_fea,"MSE：", last_mse)
            print("剔除特征后的features",feature_li)
            #计算测试集MSE
#          X_in = X_train[:,feature_li]
#          reg = LinearRegression()
#          reg.fit(X_in, y_train)
#          y_pred = reg.predict(X_test[:,feature_li])
#          MSE_test = mean_squared_error(y_test,y_pred)
#          print("第"+str(j+1)+"轮测试MSE：",MSE_test)
            #else:
                #break
print("挑选后的特征集合：",feature_li)
print("挑选特征后的训练集MSE：",last_mse)
#计算测试集MSE
X_in = X_train[:,feature_li]
reg = LinearRegression()
reg.fit(X_in, y_train)
y_pred = reg.predict(X_test)
MSE_test = mean_squared_error(y_test, y_pred)
print("后向选择测试集的MSE：",MSE_test)
```

结果

```
所有特征[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]都进行训练模型的MSE误差为：22.34005799215287
挑选后的特征集合： [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
挑选特征后的训练集MSE： 22.34005799215287
后向选择测试集的MSE： 22.098694827098132
```

在梯度下降法中修改SimpleLinearRegression中的代码，使得在fit函数中能显示每一次迭代得到的参数的MSE误差是多少，并使用load_diabetes数据集。

```
from sklearn.datasets import load_diabetes
```

代码

```
class SimpleLinearRegression:

    def __init__(self):
        self.theta = None

    def fit(self, X_train, y_train, learning_rate=0.02, n_iters=1e5, epsilon=1e-8):
        #计算目标函数
        def J(theta):
            return 1/2.0*np.sum((np.dot(X_train, theta) - y_train)**2)

        #计算梯度
        def dJ(theta):
            return np.dot(X_train.T, np.dot(X_train, theta) - y_train)

        X_train = np.hstack((X_train, np.ones((len(X_train),1))))
        last_theta = np.ones((X_train.shape[1]))
        learning_rate = learning_rate/np.mean(np.square(X_train), axis=0) # 统一量纲。也可以对数据先进行归一化
        cur_iter = 0
        while cur_iter < n_iters:
            gradient = dJ(last_theta) # 计算梯度
            theta = last_theta - learning_rate * gradient # 更新参数
            #每一次迭代得到的参数的MSE误差
            y_pred_train = np.dot(X_train, theta)
            MSE_train = mean_squared_error(y_train, y_pred_train)
            print(f'第{cur_iter}次迭代得到的参数的MSE误差为{MSE_train}')

            if (abs(J(theta) - J(last_theta)) < epsilon):  #目标函数更新小于一个足够小的值，则迭代结束
                break
```

```python
            last_theta = theta
            cur_iter += 1
        self.theta = theta

    def predict(self, x_predict):
        x_predict = np.hstack((x_predict,np.ones((len(x_predict),1))))
        return np.dot(x_predict,self.theta)
from sklearn.datasets import load_diabetes
X,y = load_diabetes(return_X_y=True)
X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=0)

reg = SimpleLinearRegression()
reg.fit(X_train, y_train, learning_rate=0.0005, n_iters=1e5, epsilon=1e-8)
print(reg.theta)

y_pred = reg.predict(X_test)
MSE_test = mean_squared_error(y_test,y_pred)
print("测试集MSE:",MSE_test)
```

结果

```
第6516次迭代得到的参数的MSE误差为2780.062316079328
第6517次迭代得到的参数的MSE误差为2780.062316079265
第6518次迭代得到的参数的MSE误差为2780.062316079203
第6519次迭代得到的参数的MSE误差为2780.062316079141
第6520次迭代得到的参数的MSE误差为2780.0623160790788
第6521次迭代得到的参数的MSE误差为2780.062316079017
第6522次迭代得到的参数的MSE误差为2780.062316078955
第6523次迭代得到的参数的MSE误差为2780.0623160788937
第6524次迭代得到的参数的MSE误差为2780.0623160788323
第6525次迭代得到的参数的MSE误差为2780.0623160787713
第6526次迭代得到的参数的MSE误差为2780.0623160787104
第6527次迭代得到的参数的MSE误差为2780.06231607865
第6528次迭代得到的参数的MSE误差为2780.0623160785894
第6529次迭代得到的参数的MSE误差为2780.062316078529
第6530次迭代得到的参数的MSE误差为2780.0623160784685
[ -43.26757207 -208.67038605  593.39834169  302.89787323 -560.25542702
  261.45981914   -8.8434348   135.93371642  703.21860088   28.34852791
  153.06797918]
测试集MSE: 3180.200468307377
```