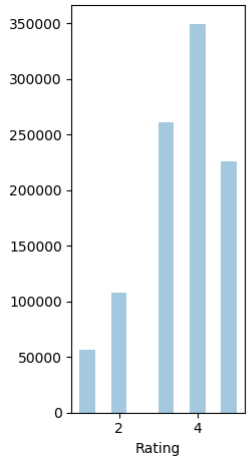
实验内容

1. 采用二分网络模型，对 ml-1m 文件夹中的“用户---电影”打分数据进行建模，考虑将用户信息、电影详细信息、以及打分分值作为该网络上的边、点的权重；
2. 根据网络结构特征给出节点相似性度量指标；
3. 基于相似性在二分网络上进行链路预测；
4. 采用交叉验证的方法验证预测结果；
5. 画出 ROC 曲线来度量预测方法的准确性。

分析与设计

- 首先观察数据，进行数据预处理
    - 通过观察数据，发现大部分人对电影的打分在3分及以下的人数为424918，大于3分的为575268人。所以假设打分>3分为喜欢，<=3分为不喜欢，尽可能减少因为数据比例对分类器准确率的影响。现在的**目标是根据用户过往打分数据预测对于给定样例用户是否会喜欢。**



    - 将用户性别标签化，0表示女，1表示男
- **转换源数据**，构建用户信息和喜欢的电影类型的特征表。每个数据特征包括用户的年龄，性别，职业，以及18个电影类型的独热码(但是用实际打分代替本应为1的地方)，标签为0表示不喜欢，1表示喜欢
- **分割**转换后的**数据集**，将数据集分成训练集，验证集，和测试集，比例为6:2:2。由于样本数量较多，如果使用KNN则每次判断时间复杂度都较大。观察电影的类别，发现两个类别之间可能存在联系，比如动作类和冒险类，所以二者可能不是相互独立的。因此，选择决策树进行分类。
- 由于样本数量较多，考虑调整分类的最小样本数量，采用枚举方式尝试min_samples_split的值。利用验证集得出较好的一个参数。因此自己**实现决策树算法**时的参数包括(dataSet, labels,min_samples_split=1)，实现时整体思路为
    - 整体采用递归结构，首部设置的**递归结束条件**包括
        - 如果待分数据集长度小于min_samples_split，则采用对数票决，多数票决的结果为正例的占比
        - 如果没有更多的特征可以使用，那么对剩下的数据进行多数票决
        - 如果所有数据的的标签相同，那么直接返回该标签(若为1，则返回1.0 否则返回0.0 代表是正例的概率)
    - 如果不满足头部条件，则首先**选择一个最优的分类属性。**我使用的是熵增益来衡量不纯度，因为我认为每个特征都是多值属性，熵增益不会产生较大偏差，但是可以减少计算量。

        - 首先计算数据集整体的熵
        - 然后循环每一个特征，计算以该特征划分数据集后的熵，计算熵增益
        - 选择熵增益最大的返回
    - **将数据集以最优属性划分**，保存为字典的形式，递归的用分割后的子集构建子树
- 将训练好的模型在测试集上进行测试
    - 根据我的设计，分类结果为概率值，我以0.5为分界，将值映射到{0,1}
    - 之后，与测试集的标签对比得到准确率

详细实现

数据预处理

```
users_cols = ["UserID", "Gender", "Age", "Occupation", "Zip-code"]
    users =pd.read_table('./ml-1m/users.dat',sep="::",names = users_cols,
engine='python')
    users.loc[users["Gender"]=="F","Gender"] = 0 # 手动进行标签化，0表示F，1表示M
    users.loc[users["Gender"]=="M","Gender"] = 1
```

分割数据集

```
################################### 首先对源数据进行转换
#################################
def get_genre_names(movies):
    # 对每条打分信息的电影进行独热编码，得到电影类型的稀疏矩阵
    genres = movies["Genres"]
    genres_lst = []
    for i in range(len(genres)):
        genres_lst += genres[i].split("|")
    genres_lst =["MovieID"] + list(set(genres_lst))
    df = pd.DataFrame(columns= genres_lst)
    df["MovieID"] = movies["MovieID"].values
    s = pd.Series()
    for i in range(len(genres)):
        lst = genres[i].split("|")
        for j in range(len(lst)):
            df.loc[i,lst[j]]= 1
    return df.fillna(0)
def fill_users_gerners(users,X_train,path,movies =
pd.read_csv("movies_genres.csv")):
    # 将每条打分信息中的用户信息和对电影类型的打分数据构建成新表
    movie_name = ['Mystery', 'Film-Noir', 'Fantasy', 'Comedy', "Children's",
'Drama',
                  'Romance', 'Horror', 'Thriller', 'Sci-Fi', 'Documentary',
'Action',
                  'Animation', 'War', 'Crime', 'Musical', 'Adventure',
'Western']
    feature = pd.DataFrame(np.zeros((len(X_train), 22)),
columns=list(users.columns.values)+movie_name) # 构建一个由用户信息和电影分类构成的空
df
    for i in range(len(X_train)):
        u_id = X_train.loc[i,"UserID"]
        m_id = X_train.loc[i,"MovieID"]          # 此处的movies为上一个函数的结果
```

```
        feature.loc[i,"UserID":"Occupation"] = users.loc[u_id-
1,"UserID":"Occupation"]
        feature.loc[i, "Mystery":"Western"] =
movies.loc[movies["MovieID"]==m_id,"Mystery":"Western"].values
    feature.to_csv(path,encoding="gbk",index=False)
################################## 按6:2:2 分割训练集，验证集，测试集
#######################
    users,movies,ratings= read_data()
    label = ratings["Rating"]
    label[label <= 3] = 0  # 为了让分割结果尽可能均匀，先进行标签化
    label[label > 3] = 1
    feature = pd.read_csv("./feature2.csv").drop("UserID",axis=1).astype(int)
    X_tt, X_validation, Y_tt, Y_validation = train_test_split(feature, label,
test_size=0.2)  # 返回测试集，验证集，测试集标签，验证集标签
    X_train, X_test, Y_train, Y_test = train_test_split(X_tt, Y_tt,
test_size=0.25)
```

实现决策树

计算熵

```
def calc_shannon_ent(dataSet): # 计算信息熵
    num_entries = len(dataSet)  # 返回数据集的行数
    label_counts = {} # 保存每个标签出现的次数
    # 为所有可能分类创建字典
    for featVec in dataSet:# 对每组特征向量进行统计
        current_label = featVec[-1] # 提取标签信息
        if current_label not in label_counts.keys():# 如果标签没有放入统计次数的字
典，则加入
            label_counts[current_label] = 0
        label_counts[current_label] += 1# 标签计数
    shannoent = 0.0
    # 以二为底求对数
    for key in label_counts:
        prob = float(label_counts[key])/num_entries
        shannoent -= prob * math.log(prob, 2)
    return shannoent
```

取出第axis列等于value的特征

```
def split_dataset(dataSet, axis, value):# 取出第axis列等于value的特征
    tmp = dataSet[dataSet[:,axis]==value]
    return np.delete(tmp,axis,axis=1)
```

求最优划分属性

```
def choose_best(dataSet): # 按照信息增益，计算最好的分类标准
    numFeatures = len(dataSet[0]) - 1 # 减去标签
    baseEntropy = calc_shannon_ent(dataSet) # 整体的熵
    bestInfoGain = 0.0
    bestFeature = -1

    # 创建唯一分类标签
    for i in range(numFeatures):
        featList = list(dataSet[:,i])
```

```
        uniqueVals = set(featList)
        newEntropy = 0.0

        # 计划每种划分的信息熵
        for value in uniqueVals: # 循环每种取值
            subDataSet = split_dataset(dataSet, i ,value) # 对于第i类取值为
            prob = len(subDataSet)/float(len(dataSet))
            newEntropy += prob * calc_shannon_ent(subDataSet)
            infoGain = baseEntropy - newEntropy

            # 计算最好的增益熵
            if infoGain > bestInfoGain:
                bestInfoGain = infoGain
                bestFeature = i

    return bestFeature
```

多数票决过程 返回正例占比

```
def majoritycnt(classList): # 实现多数票决函数
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
        classCount[vote] += 1
    # return max(classCount,key=classCount.get)# 返回最大的值对应的键
    return has_key(classCount,1)/len(classList)
```

构建树/训练模型 (主过程)

```
def create_tree(dataSet, labels,min_samples_split=1): # 创建决策树
    classList = list(dataSet[:,-1]) # 得到所有的标签
    if len(dataSet)<=min_samples_split:
        return majoritycnt(classList)
    if len(dataSet[0]) == 1:
        # 停止分割直至没有更多特征，则多数票决
        return majoritycnt(classList)
    if classList.count(classList[0]) == len(classList):
        # 停止分类直至所有类别相等,都是classList[0]
        return 1.0 if classList[0] == 1 else 0.0

    bestfaet = choose_best(dataSet)# 根据信息熵计算最优标签
    # print(labels,bestfaet)
    bestfaetlabel = labels[bestfaet]
    mytree = {bestfaetlabel:{}} # 根据最有特征的标签生成树
    del(labels[bestfaet])# 删除已经使用的标签
    # 得到包含所有属性的列表
    featvalues = list(dataSet[:,bestfaet])
    uniquevalues = set(featvalues) # 去掉重复的属性
    for value in uniquevalues: # 遍历特征创建决策树
        sublables = labels[:]
        mytree[bestfaetlabel][value] = create_tree(split_dataset(dataSet,
bestfaet, value), sublables,min_samples_split)

    return mytree
```

测试模型

预测函数

```
def classify(inputtree, featlabels, testvec):# 这里与构建树类似，也是递归进行，不在赘述
    firststr = list(inputtree.keys())[0]
    seconddict = inputtree[firststr]
    featindex = featlabels.index(firststr)
    key = testvec[featindex]
    if key not in seconddict.keys(): # 如果firststr没有取值为key的，那么随机一个打分
        valueoffeat = np.random.rand()
    else:
        valueoffeat = seconddict[key]
    if isinstance(valueoffeat, dict):# 没到叶子，继续划分
        classlabel = classify(valueoffeat, featlabels, testvec)
    else:
        classlabel = valueoffeat# 叶子结点，返回打分
    return classlabel
```

测试模型，画出ROC曲线

```
def test_tree(inputtree, featlabels, testdata,Y_test):
    prob = np.array([classify(inputtree, featlabels,testdata[i]) for i in
range(len(testdata))]).reshape(-1,1)
    pred =np.array([1 if i>0.5 else 0 for i in prob]).reshape(-1,1)
    acc = (pred==Y_test).sum()/len(Y_test)
    print("-ACC",acc)
    return acc
    print(prob)
    fpr, tpr, threshold = roc_curve(Y_test, prob)  # 计算真正率和假正率
    roc_auc = auc(fpr, tpr)  # 计算auc的值
    plt.figure()
    lw = 2
    plt.figure(figsize=(10, 10))
    plt.plot(fpr, tpr, color='darkorange',
             lw=2*lw, label='ROC curve (area = %0.3f)' % roc_auc)  ###假正率为横坐
标，真正率为纵坐标做曲线
    plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.0])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()
```

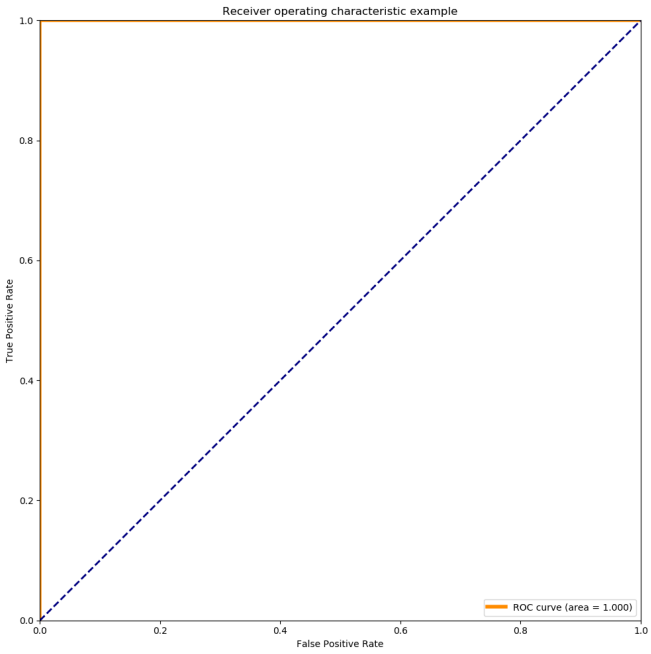
十折交叉验证

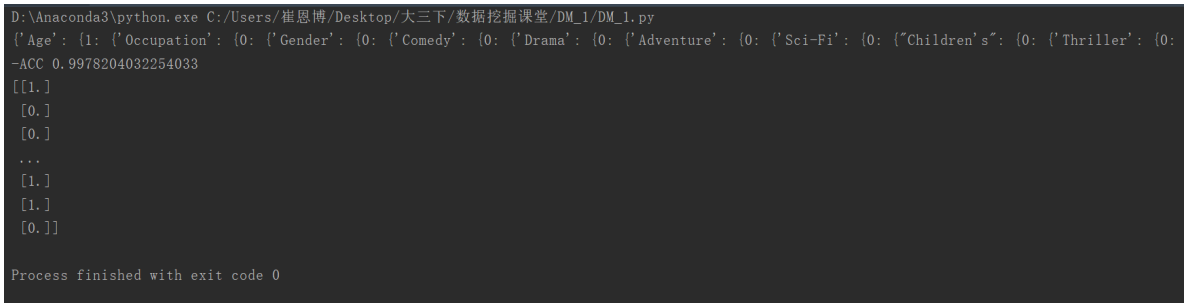
```
    X = [i for i in range(10)]
    Y = [i for i in range(10)]

    X_tt, X[9], Y_tt, Y[9] = train_test_split(feature, label, test_size=1 / 10)
    X_tt, X[8], Y_tt, Y[8] = train_test_split(X_tt, Y_tt, test_size=1 / 9)
    X_tt, X[7], Y_tt, Y[7] = train_test_split(X_tt, Y_tt, test_size=1 / 8)
    X_tt, X[6], Y_tt, Y[6] = train_test_split(X_tt, Y_tt, test_size=1 / 7)
```

```
    X_tt, X[5], Y_tt, Y[5] = train_test_split(X_tt, Y_tt, test_size=1 / 6)
    X_tt, X[4], Y_tt, Y[4] = train_test_split(X_tt, Y_tt, test_size=1 / 5)
    X_tt, X[3], Y_tt, Y[3] = train_test_split(X_tt, Y_tt, test_size=1 / 4)
    X_tt, X[2], Y_tt, Y[2] = train_test_split(X_tt, Y_tt, test_size=1 / 3)
    X[0], X[1], Y[0], Y[1] = train_test_split(X_tt, Y_tt, test_size=1 / 2)
    feature = ['Gender', 'Age', 'Occupation', 'Mystery', 'Film-Noir',
       'Fantasy', 'Comedy', "Children's", 'Drama', 'Romance', 'Horror',
       'Thriller', 'Sci-Fi', 'Documentary', 'Action', 'Animation', 'War',
       'Crime', 'Musical', 'Adventure', 'Western']
    ACC = []
    for i in range(10):
        columns = feature.copy()
        print("Round",i)
        X_train = X_test = pd.DataFrame(columns=columns)
        Y_train = Y_test =  pd.Series()
        for j in range(10):
            if i!=j:
                X_train = X_train.append(X[j])
                Y_train = Y_train.append(Y[j])
            else:
                X_test = X[j]
                Y_test = Y[j]
        # print(X_train.shape,Y_train.shape)
        # print(X_test.shape,Y_test.shape)
        # clf = classifier2(X_train,Y_train)
        # test_model2(clf, X_test, Y_test)
        trainSet = np.hstack((X_train.values,Y_train.values.reshape(-1,1)))
        DecTree = create_tree(trainSet, columns, 5)
        print(DecTree)
        ACC.append(test_tree(DecTree, feature, X_test.values,
Y_test.values.reshape(-1,1)))
    print("Avg_Acc = ",np.mean(ACC))
```

实验结果

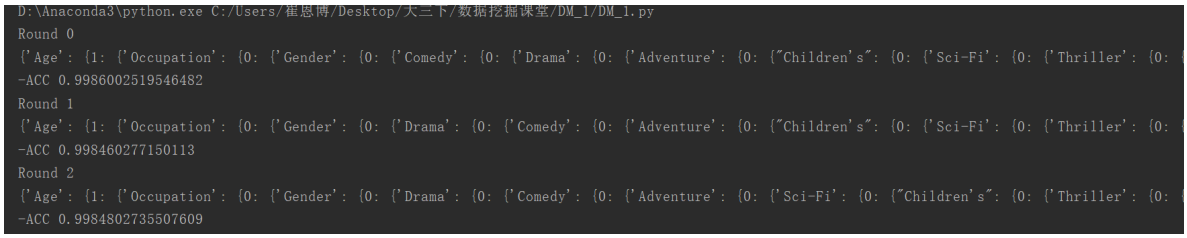
测试集上结果

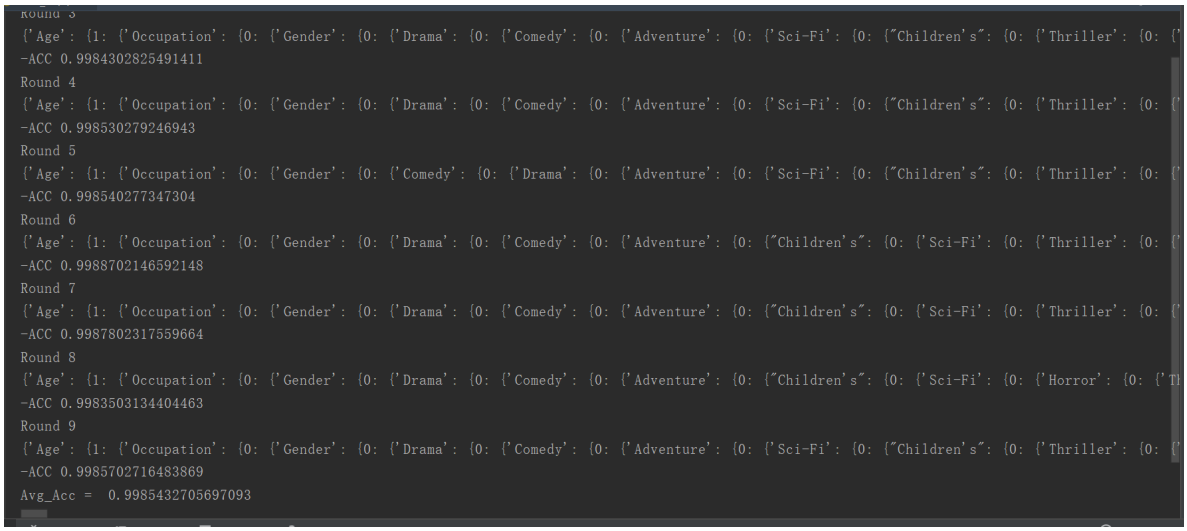




准确率为99.7%

交叉验证准确率



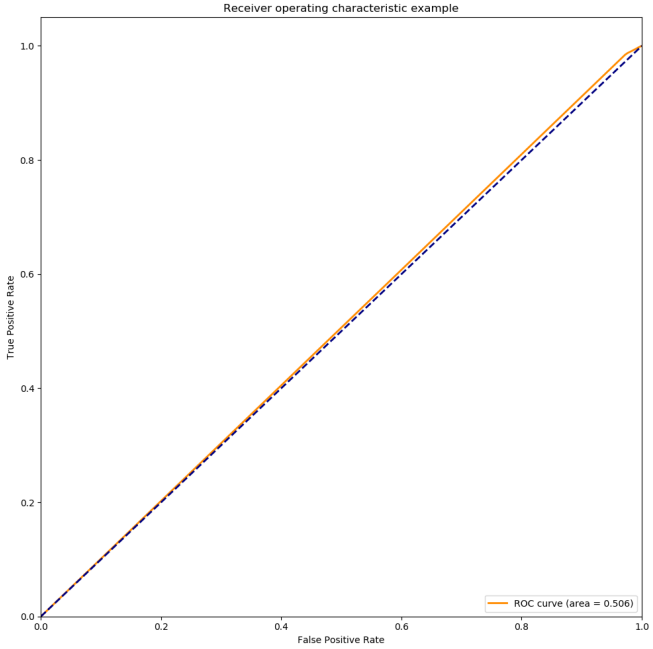


结果总结

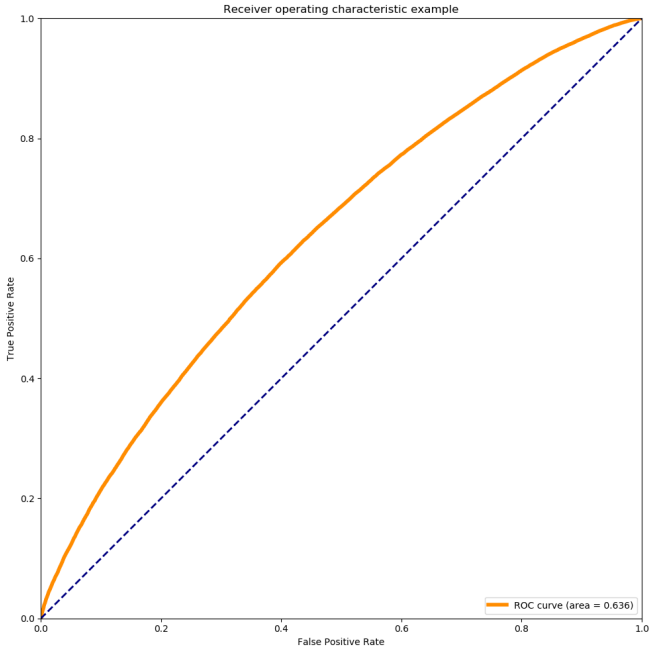
- 经过多次尝试，目前的AUC值达到1，感到很意外。十折交叉验证的准确率为99.8%
- 第一次交叉验证时，没有在划分数据集之前就将打分二值化。导致训练集和测试集的样本分布不一致，在第5.6轮开始准确率突然下降。修改之后解决问题
- 如果结果没有问题，那么本次实验让我深刻感受到决策树的强大
- 我认为我这个树构建之后和KNN的思想比较类似。

心得体会

- 第一次进行数据挖掘，感觉无从下手。转换源数据集时进行了多次尝试，第一次是将打分数据按照用户编号聚合，计算用户对每个类型的打分平均值。以用户信息为特征，每个类型的打分为标签，训练18个二分类器。对于测试样本，首先得到该电影包含哪些类型，然后对该用户使用相应类型的二分类器，得到若干个0或1表示该用户对该电影中哪些类型喜欢，或不喜欢。最后多数票决判定对该电影整体是否喜欢。但是结果很差
    - 我认为原因是特征太少，按用户编号聚合后，样本也少，导致模型预测能力很差



- 还尝试了使用用户信息和电影类型的独热码来训练模型，结果好了一些



    - 这里使用的特征只能说明用户看了这些电影类型，而没有将是否喜欢(具体打分)考虑进去
    - 所以最后用用户给每个类型的打分和用户信息为特征来训练模型，得到实验结果中的ROC曲线
- 遇到过的问题：起初，我按照直觉将大于等于3的都认为是正例，导致最后准确率很高(83%),但是AUC很小(0.5)。后来才发现我的准确率高是因为样本分布的问题，实际模型并不可靠。本次实验令我深刻理解了ROC曲线的含义。准确率可能会受到数据分布的影响，但是ROC曲线能较好的反应模型的可信度。
- 我认为准确率高是因为训练样本足够多，比较训练集的结果和交叉验证结果可以看出随着训练样本的增多，准确率还在提高。如果避开所有训练样例中每个属性取值，那么预测结果就是完全随机的打分。这也说明了数据的重要性