机器学习与数据挖掘大作业

多标签用户人格分类（集成学习）

数据科学与计算机学院　17大数据与人工智能

17341015　陈鸿峥

目录

一、 题目描述

MBTI理论认为一个人的个性可以从四个角度进行分析，用字母代表如下：

- 驱动力的来源：外向E—内向I
- 接受信息的方式：感觉S—直觉N
- 决策的方式：思维T—情感F
- 对待不确定性的态度：判断J—知觉P

按照不同的组合，可以产生16种人格类型。

本次大作业要求利用机器学习方法，通过用户的发言记录对用户的人格类型进行分类[1]。

本实验报告为大作业的第一部分—**集成学习**方法。

二、 预处理

由于原始数据集为用户在网页上的发言记录，非常随意及杂乱，故做分类任务前的第一步需要灵活应用上学期自然语言处理学过的知识，对数据集进行预处理操作。主要分为文本清洗、词典生成、词向量生成三个步骤。

1. 文本清洗

文本清洗一步依序进行了以下操作。

1. 标点后强制添加空格。由于是网页发言记录，很多用户可能不会太过在意标点符号后面是否有空格。如果没有添加，则之后删除标点可能会导致前后两个词语粘连。
2. 移除网页链接。由于网页地址通常没有太多含义，故这里直接将发言记录中出现的网页链接直接移除。采用以下的正则表达式对http/https开头的网址进行匹配。

```
(http|https):\/\/.*?( |'|\")
```

3. 移除数字。同理，数字带来的信息非常少，无法作为区分人格的合理因素，故也将其移除。
4. 移除标点符号。我们做文本分类时，关心的是用户说过的词语，而不是他是否有进行断句，故也将所有标点符号移除。这里采用了一种比较高效的方法，直接调用C库对字符串进行处理。

```
new_post = new_post.translate(str.maketrans('', '', string.punctuation))
```

注意两条评论之间采用|||进行分隔，但我们直接将其移除了，意味着我们并不关心用户发言记录的先后顺序，我们更关心的其说过什么话。

5. 移除空格。句子内部空格用正则表达式匹配" +"后用单一空格代替；句子前后的空格用Python内置的strip函数即可移除。

[1]数据集链接：https://www.kaggle.com/datasnaek/mbti-type

6. 字母小写化。避免计算机认为不同大小写字母构成的为不同的词语，如"Hello"和"hello"两者意思相同，代表的是同一个词，故要进行规范化。
7. 移除停止词(stopping words)。这是很多NLP任务中必备的一步，因为停止词并不会带来任何更多语义上的信息。这里采用上学期NLP大作业提供的停止词列表 [1]。

做完上述预处理后，即可将处理后的用户发言存成文档/列表，文档中的每一行或列表中的每一项即为对应用户所说过的所有词语。

这里做的操作都比较常规，并没有对该数据集进行特定优化，如果后续作业中想继续对预处理部分进行改进，可以考虑以下几个方面：

- 表情符号的处理。从这些用户的发言记录中可以发现，外国网友也是很喜欢用emoji甚至颜文字的。 emoji比较好辨别，均用两个冒号括起来，如`:sad:`。颜文字如`:)`和`:(`，这是用得比较多的，但被我们在前面的预处理中移除了。而喜欢用这些表情符号的可能恰恰对应着某一类人，因此可以单独对其进行处理，做成单独的特征，供后续的机器学习模型使用。
- 词型变位。目前还没有做这方面的处理，可能需要结合nltk包将不同的词性但代表同一个词语的单词进行合并。这可能会对后续词频及词向量的生成造成一定影响。

2. 生成词典

由于机器学习任务通常都是输入数字值，因此需要将单词全部转为数字标号，以便之后更好送入机器学习模型。

主要的操作有以下几步：

1. 构造词表。将所有用户所说过的词语合并起来即构成词表。
2. 词频计数。将Python标准库中的`Counter`作用在词表上，即可得到每个词语的词频。
3. 添加未知词标记。 NLP任务中常常有未登录词(Out-of-vocabulary, OOV)的情况出现，即在测试集中出现的词语可能并未在训练集中出现。因此需要添加`<UNK>`标记，并将其标号为0，以更好处理这种情况。（之后采用更强大的机器学习模型，可能还需要对句子进行对齐处理，这时还需要添加`<PAD>`标记）
4. 删除词频过小的词语。词频过小的词语对预测任务并没有带来太多实质性的帮助，同时还会导致特征空间维度过大，因此需要提前将这些词语给删除。
5. 依词频排序。对遗留的词语依照词频降序排序。
6. 基于词频生成词嵌入。通过读取排序的序号，即可生成词语到数字标号的双向映射。

完整过程如下面程序所示。

```
1        word_counts = Counter(word_lst)
2        word_counts["<UNK>"] = max(word_counts.values()) + 1
```

```
    # remove words that don't occur too frequently
    print("# of words before:",len(word_counts))
    for word in list(word_counts): # avoid changing size
        if word_counts[word] < 6:
            del word_counts[word]
    print("# of words after:",len(word_counts))
    # sort based on counts, but only remain the word strings
    sorted_vocab = sorted(word_counts, key=word_counts.get, reverse=True)

    # make embedding based on the occurance frequency of the words
    int_to_word = {k: w for k, w in enumerate(sorted_vocab)}
    word_to_int = {w: k for k, w in int_to_word.items()}
```

3. 生成词向量

由于第一次任务较为简单，故这里直接使用词袋模型(Bag of Words, BoW) [2]构造词向量。

考虑词典中所有词语为$\{w_i\}_{i=1}^{M}$，其中M为词典大小。那么对于每一用户uid的发言，可构造词向量

$$\mathbf{v}_{uid} = \begin{bmatrix} \mathcal{W}_{uid}(w_1) & \mathcal{W}_{uid}(w_2) & \cdots & \mathcal{W}_{uid}(w_M) \end{bmatrix}^{\mathrm{T}}$$

其中$\mathcal{W}_{uid}(w_i)$计算用户uid中w_i出现的次数。这里同样可以直接通过Python标准库的`Counter`类进行计数。注意这里需要利用前面构造的词典将字符串单词转换为数字标号，如果词典中该词没有出现，则标号为0`<UNK>`。

最终构造出来的词袋模型（也即送入机器学习模型的特征）为$N \times M$大小的矩阵，其中N为用户数目。

三、 集成模型

下面将介绍构造集成模型并进行训练预测的完整过程。这里我采用了随机森林作为集成模型，基学习器为决策树。

1. 决策树

在本次实验中，我自己实现了ID3决策树，以信息增益作为决策准测，并添加了预剪枝的机制。

(i) 熵与信息增益

依据信息熵的定义，假定当前样本集合D中第k类样本所占的比例为$p_k (k = 1, 2, \ldots, |\mathcal{Y}|)$，则$D$的信息熵为

$$\mathrm{Ent}(D) = -\sum_{k=1}^{|\mathcal{Y}|} p_k \log_2 p_k$$

又假定离散属性a有V个可能的取值$\{a^1, a^2, \ldots, a^V\}$，若使用$a$来对样本集$D$进行划分，则会产生$V$个分支结点，其中第$v$个分支结点包含了$D$中所有在属性$a$上取值为$a^v$的样本，记为$D^v$，进而可计算出信息增益

$$\mathrm{Gain}(D, a) = \mathrm{Ent}(D) - \sum_{v=1}^{V} \frac{|D^v|}{|D|} \mathrm{Ent}(D^v)$$

为了高效实现信息熵和信息增益的计算，我并没有采用Python的裸循环实现，而是利用NumPy的向量化计算方法。对照上述公式实现可得到以下代码。

```
def entropy(p):
    """
    Input: p (prob) is a numpy array
    Output: Ent = - \sum_i p_i \log p_i
    """
    if p.ndim == 1:
        new_p = p[p != 0]
        return -np.sum(new_p * np.log2(new_p))
    else: # high dimensional input (should be guaranteed no zeros exist)
        return -np.sum(p * np.log2(p),axis=1)

def information_gain(D,a,L):
    """
    Input: D (dataset), a (attribute), L (labels)
        attributes are features, and all discrete
    Output: Gain = Ent(D) - \sum_v |D_v|/|D| Ent(D_v)
        where v\in V is the unique values of a
    """
    _, pk = value_counts(L,normalize=True)
    V, prop_Dv = value_counts(D[:,a],normalize=True) # proportion = |D_v|/|D|
    # prob (|V|,|class|)
    sumup = 0
    for av, prop in zip(V,prop_Dv):
        _, prob_Dv = value_counts(L[D[:,a] == av],normalize=True)
        sumup += prop * entropy(prob_Dv)
    return (entropy(pk) - sumup, a)
```

(ii) 树结点

决策树最基本的单元即树的结点，这里我采用一个Node类进行封装。其中可以设置是叶子(leaf)结点还是分支(branch)结点，并用字典存储其孩子对象，方便后续的访问及遍历。

```

class Node:
    """
```

```
    Class Node in decision tree
    """

    def __init__(self):
        self.branch = {}

    def setLeaf(self,catagory,cnt=1):
        """
        Set this node as a leaf with "catagory"
        """
        self.label = "Leaf"
        self.catagory = catagory

    def setBranch(self,attr,value,node):
        """
        Set this node as a parent node with "attr"
        Add a child "node" with "value"
        """
        self.label = "Branch"
        self.attr = attr
```

(iii) 树构造

依照西瓜书中给出的决策树算法，可以对应写出`TreeGenerate`函数。

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
　　属性集 $A = \{a_1, a_2, \ldots, a_d\}$.
过程： 函数 TreeGenerate(D, A)
1: 生成结点 node;
2: **if** D 中样本全属于同一类别 C **then**
3: 　将 node 标记为 C 类叶结点; **return**
4: **end if**
5: **if** $A = \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
6: 　将 node 标记为叶结点, 其类别标记为 D 中样本数最多的类; **return**
7: **end if**
8: 从 A 中选择最优划分属性 a_*;
9: **for** a_* 的每一个值 a_*^v **do**
10: 　为 node 生成一个分支; 令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集;
11: 　**if** D_v 为空 **then**
12: 　　将分支结点标记为叶结点, 其类别标记为 D 中样本最多的类; **return**
13: 　**else**
14: 　　以 TreeGenerate(D_v, $A \setminus \{a_*\}$)为分支结点
15: 　**end if**
16: **end for**
输出： 以 node 为根结点的一棵决策树

图 4.2 决策树学习基本算法

下列代码中已经将上述的三种情况全部标注出来，详情见代码中的注释。为了提升泛化能

力，这里还提供了预剪枝(pre-pruning)机制。

```
    def TreeGenerate(self,dataset,labels,attributes,depth,
                     cnt_leaves=0,root=None,prev_best=None):
        """
        Core algorithm of ID3 that generates the whole tree
        """
        catagory = np.unique(labels)
        node = Node() if root == None else root # better used for validation
            ↪ indexing
        cnt_leaves += 1

        # 1) All samples in "dataset" belongs to the same catagory
        if len(catagory) == 1:
            node.setLeaf(catagory[0],cnt_leaves)
            return node

        # 2) "attributes" is empty, or the values of "dataset" on "attributes" are
            ↪ the same
        if len(attributes) == 0 or       np.array([len(np.unique(dataset[:,a])) ==
            ↪ 1 for a in attributes]).all() == True:
            node.setLeaf(mode(labels),cnt_leaves)
            return node

        """The general case"""
        # find the attribute with greatest information gain
        max_gain = (-0x3f3f3f3f,None)
        if self.random:
            k = int(np.log2(len(attributes))) + 1 # random set!!!
        else:
            k = len(attributes)
        random_attributes = attributes[np.random.choice(len(attributes),k,replace=
            ↪ False)]
        for a in random_attributes:
            gain = information_gain(dataset,a,labels)
            if gain[0] > max_gain[0]:
                a_best, max_gain = a, gain
        unique = self.unique_val[a_best] # be careful, not dataset!
        num_leaves = len(unique)
        print("Tree {} Depth {}: {} - {}\t # leaves: {}".format(self.tid, depth,
            ↪ prev_best, max_gain, num_leaves))

        if self.prune and self.test_set != None:
            # without partition
            node.setLeaf(mode(labels),cnt_leaves)
            acc_without_partition = self.test(val=True)
```

```
40
41          # with partition (make branches)
42          for av in unique:
43              Dv = dataset[dataset[:,a_best] == av,:]
44              labels_v = labels[dataset[:,a_best] == av]
45              cnt_leaves += 1
46              leafnode = Node()
47              if len(Dv) == 0:
48                  leafnode.setLeaf(mode(labels),cnt_leaves)
49              else:
50                  leafnode.setLeaf(mode(labels_v),cnt_leaves)
51              node.setBranch(a_best,av,leafnode)
52          print("Depth {} ({}): ".format(depth,cnt_leaves))
53          acc_with_partition = self.test(val=True,print_flag=True)
54
55          # pre-pruning (to make sure it has generated sufficient nodes, depth is
                ↪  set here)
56          if depth > 5 and acc_without_partition >= acc_with_partition:
57              cnt_leaves -= num_leaves
58              print("Prune at {}: {} (without) >= {} (with)".format(a_best,
                    ↪ acc_without_partition,acc_with_partition))
59              node.setLeaf(mode(labels))
60              return node
61          elif depth > 5:
62              print(a_best,acc_without_partition,acc_with_partition)
63
64          # true partition (branching makes more gains)
65          for av in unique:
66              Dv = dataset[dataset[:,a_best] == av,:]
67              labels_v = labels[dataset[:,a_best] == av]
68              # 3) "Dv" is empty, which can not be partitioned
69              if len(Dv) != 0:
70                  node.setBranch(a_best,av,self.TreeGenerate(Dv,labels_v,
71                                                             attributes[attributes !=
                                                                 ↪ a_best],
72                                                             depth+1,cnt_leaves))
73      else:
74          if depth > self.max_depth:
75              node.setLeaf(mode(labels),cnt_leaves)
76              return node
77          for av in unique:
78              Dv = dataset[dataset[:,a_best] == av,:]
79              labels_v = labels[dataset[:,a_best] == av]
80              cnt_leaves += 1
81              leafnode = Node()
```

```
82              if len(Dv) == 0:
83                  leafnode.setLeaf(mode(labels),cnt_leaves)
84                  node.setBranch(a_best,av,leafnode)
85              else:
86                  node.setBranch(a_best,av,self.TreeGenerate(Dv,labels_v,
87                                                             attributes[attributes !=
                                                                   ↪ a_best],
88                                                             depth+1,cnt_leaves,
                                                                   ↪ prev_best=a_best))
89      return node
```

从代码中可以看到，对特征数据的处理都尽可能采用NumPy的向量化方式，及fancy indexing进行索引，这样能够最大程度提升性能，避免不必要的解释运行开销。

另外，在代码第24行引入了随机化特征选择，这会在下面的部分进行叙述。

2. 随机森林

随机森林如算法1所示。这里有几个关键点：

- 在决策树特征选取过程中引入随机化因素。原来的决策树是在当前结点的所有属性值中挑选最优的一个，现在则是从当前结点所有属性值中先挑选k个，再从这k个中选择最优的一个。这里k取$\lfloor \log_2 d \rfloor + 1$，其中$d$为属性数目。
- 采用自助采样法(bootstrap sampling)对输入的训练集进行选择，同样是提升泛化能力的有效机制。
- 最终利用bagging的方式进行预测，具体实现中采用了简单投票法。

Algorithm 1 Random Forest (Bagging)

1: **for** $t = 1, 2, \ldots, T$ **do**
2: Use bootstrap sampling to generate $\mathcal{D}_{bs}$
3: $h_t = RF(\mathcal{D}_{bs})$
4: **return** $H(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{t=1}^{T} \mathbf{1}(h_t(\mathbf{x}) = y)$

完整的随机森林构造如下所示，这里参照了sklearn的API函数签名以方便后续实验对比，但里面的代码均为自己实现。

```
1  class RandomForest():
2
3      def __init__(self,n_trees=3,max_depth=5):
4          self.n_trees = n_trees
5          self.dt = [0] * self.n_trees
6          self.max_depth = max_depth
7
```

```
    def fit(self,train_X,train_Y):
        start_time = time.time()
        tasks = []
        for i in range(self.n_trees):
            def train_dt(idx):
                samp_X, samp_Y = resample(train_X,train_Y,n_samples=len(train_X))
                print("Building tree {}".format(idx))
                dt = ID3(train_set=(samp_X.astype(int), samp_Y.astype(int)),
                        attributes=np.array(list(range(train_X.shape[1]))),
                        unique_val=unique_X_val,
                        prune=False,
                        max_depth=self.max_depth,
                        tid=idx,
                        random=True)
                dt.train()
                pickle.dump(dt,open("tree-{}.pkl".format(idx),"wb"))
            p = multiprocessing.Process(target=train_dt,args=(i,))
            tasks.append(p)
        [x.start() for x in tasks]
        [x.join() for x in tasks]
        for i in range(self.n_trees):
            self.dt[i] = pickle.load(open("tree-{}.pkl".format(i),"rb"))
        print("Random forest time: {}s".format(time.time() - start_time))

    def predict(self,test_X):
        print("Generating prediction...")
        pred = np.zeros((len(test_X),))
        for i, row in enumerate(test_X):
            counts = np.zeros((len(labels),))
            for j in range(self.n_trees):
                pred_one = self.dt[j].predict(row)
                counts[pred_one] += 1
            pred[i] = np.argmax(counts)
        return pred

    def score(self,test_X,test_Y):
        pred_Y = self.predict(test_X)
        acc = np.sum(pred_Y == test_Y) / len(test_Y)
        print(classification_report(pred_Y,test_Y,target_names=labels))
        return acc
```

注意这里还使用了Python的多进程库`multiprocessing`[2]来进行并行计算。由于每棵决策

[2]之所以不使用多线程是因为Python具有全局解释器锁(Global Interpreter Lock, GIL)，在多线程执行过程中会造成非常大的性能损耗。

树的构造过程都是独立的，因此对每棵树分配一个进程即可。进程与进程之间直接采用文件进行通信。

3. 模型训练与预测

有了上述封装好的模块，要进行模型训练与预测就比较容易了，如下调用函数。

```
rf = RandomForest(n_trees=N_TREES,max_depth=MAX_DEPTH)
rf.fit(X_train,y_train)
acc = rf.score(X_test,y_test)
print("Acc: {:.2f}%".format(acc * 100))
```

4. 其他实现细节

- 只使用了基本了numpy和pandas库用于做矩阵运算及数据处理，引入sklearn库仅用于数据集划分及实验对比。
- 预处理的中间结果用NumPy或pickle文件保存。重新执行时会先判断文件是否存在，如果存在则直接读取，而不用重复进行预处理。

四、 实验结果

本次实验的完整代码可见附件中的`Ensemble.ipynb`或由Jupyter Notebook生成的`Ensemble.py`文件。

1. 超参数选择

实验所采用的超参数如表1所示。之所以要设置词典最小词频和决策树最大深度，是为了减少特征维度，进而缩短训练时间。

表 1: 超参数设置

词典中的最小词频	5
随机森林基学习器数目	1,3,5,10
决策树最大深度	10
训练集测试集比例	7:3

生成的特征矩阵维度为8676×27129，前者为用户数目，后者为词典大小。

2. 综合比较

最终各类别的F1及精确度指标如图1所示，可以看到我的随机森林最终达到了31%的准确率。

	precision	recall	f1-score	support
INFJ	0.42	0.45	0.44	449
ENTP	0.09	0.10	0.10	188
INTP	0.41	0.38	0.40	405
INTJ	0.29	0.34	0.31	303
ENTJ	0.03	0.03	0.03	65
ENFJ	0.00	0.00	0.00	62
INFP	0.47	0.53	0.50	541
ENFP	0.11	0.09	0.10	222
ISFP	0.03	0.02	0.02	90
ISTP	0.19	0.15	0.17	98
ISFJ	0.05	0.06	0.06	47
ISTJ	0.02	0.02	0.02	61
ESTP	0.00	0.00	0.00	30
ESFP	0.00	0.00	0.00	17
ESTJ	0.00	0.00	0.00	15
ESFJ	0.00	0.00	0.00	10
accuracy			0.31	2603
macro avg	0.13	0.14	0.13	2603
weighted avg	0.30	0.31	0.30	2603

图 1: 实验结果

从图中可以看出能够预测准确的基本上都是出现次数比较多的，而对于出现较少的性格基本预测不出。这其实很大程度是因为原始数据集各类别的分布不均，如图2所示。该数据集中ESTJ、ESFJ、ESFP等性格出现的次数本来就少，要从仅仅几十个样本中就学出其规律，对于人类来说都很难，更何况是机器，故可以看到这些类别的F1分数均为0。如果要处理好这种情况，则可能会涉及到小样本学习(few-shot learning)。这已经超出了本次实验的范围，故在此不做更加深入的分析。

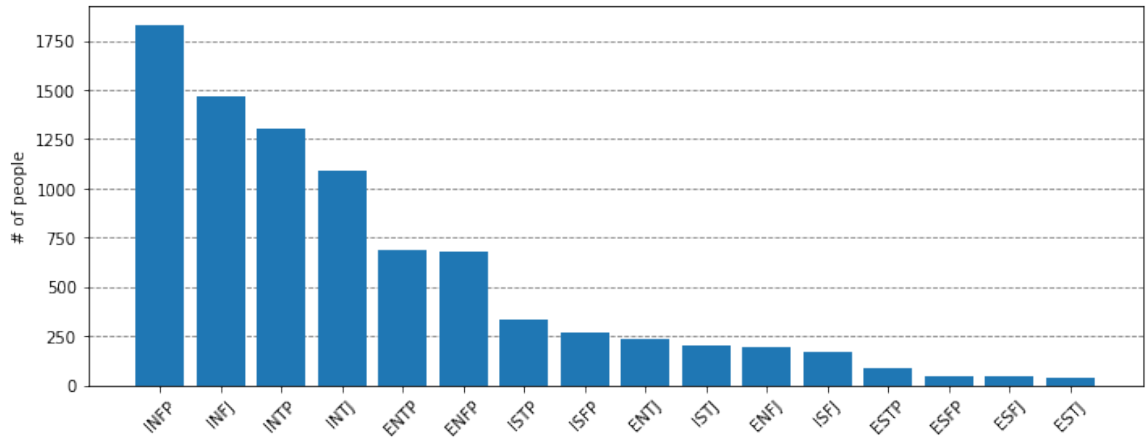

图 2: 用户个性分布

为了探究不同超参数对模型的影响，我也比较了我自己实现的随机森林与sklearn默认随机森林的结果，如图3所示。可以看到单棵决策树我的性能要比sklearn的性能高几个百分点，但随着决策树数目的增加，sklearn的效果越来越好，并超过我的随机森林的性能。这很大程度取决于自助采样的选择以及超参数随机化因子的设定。同时由于没有在决策树内部开启并行计算，我实现的版本比sklearn的版本要慢上几个数量级。

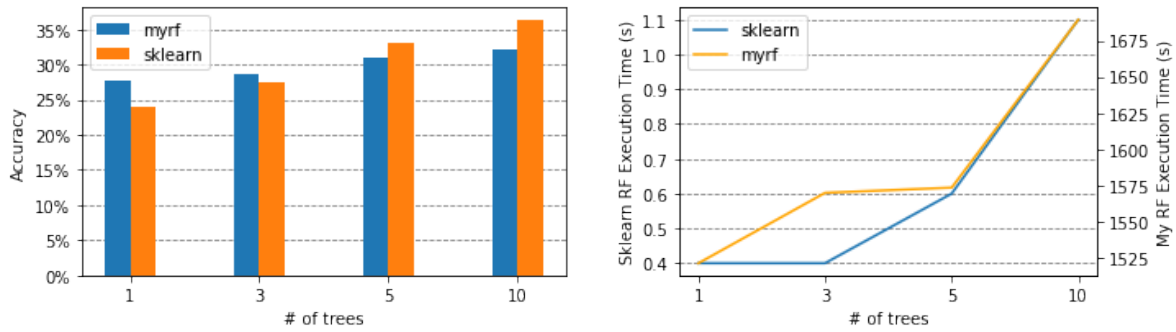

图 3: 不同超参数下准确率与运行时间比较

但总的来说，我的精度与sklearn的精度控制在合理的范围内，并且对于该数据集表现出了较好的泛化能力。

五、 总结与思考

本次实验相对来讲比较简单，对于文本数据充分利用上学期自然语言处理课程的知识来进行处理，而难点应该是在后面的手写决策树及随机森林。由于有一定的代码量，因此这一部分还是调试了挺长时间。为了验证决策树的正确性，还先利用西瓜书里面的数据集进行测试。确保没有问题后，才将其运用到此次Kaggle的数据集中。

此次实验还有很多未完善的地方，如未做特征工程（利用PCA等工具先进行降维），未采

用更好的词向量模型（如TF-IDF和word2vec），未对样本量较少的类别进行处理等，这些都可以在之后的实验中继续进行补充。另外从本次实验中也可以看出NLP相关任务的难度，即使是采用了强大的机器学习模型，但预测准确率依然比较低，不过这也给了之后的实验更大的探索空间。

参考文献

[1] Stopwords, https://github.com/goto456/stopwords

[2] Bag of words model, https://en.wikipedia.org/wiki/Bag-of-words_model