Report

学号	姓名	方向
18340013	陈琮昊	人工智能与大数据

一、Experimental Task

This experiment requires us use `Random Forest` method without related packages to classify the `MBTI` personality in the data set of `Kaggle`. The data set gives a total of 8675 personality tags and corresponding posts text data. We need to analyze the text content of posts and extract features to finish this classification task.

Personality tags are as follows:

1. Extroversion (E) - Introversion (I)
2. Sensing (S) - Intuition (N)
3. Thinking (T) - Feeling (F)
4. Judging (J) – Perceiving (P)

After the combination, we can get 16 personality types. For convenience, a string of abbreviations is used to represent a personality type. For example, `INTP` represent `'Introversion, Intuition, Thinking, Perceiving'`.

二、Experimental Principle

1. Random Forest:

1.1 Bagging:

Bagging is one of the ideas in `Ensemble Learning` . Ensemble learning is to combine multiple weak models to get a better and more robust model. The potential idea of ensemble learning is that even if one weak classifier gets the wrong prediction, other weak classifiers can correct the errors. There are two points worth noting:

First, the sample set is extracted from the original set. Each round, training samples are extracted from the original set by bootstrapping method (*i.e.* some samples may be selected many times, while may never be selected). K training samples are obtained by K rounds of extraction (K is a number). The most important, K training samples are independent of each other!

Second, each training samples is used to get a model, so K training samples are used to get K models. The K models predict the final classification results by voting. All models have the same importance (*i.e.* weight)! If each model has different weight, that's boosting not bagging.

If we combine bagging with decision tree, we get the random forest method.

1.2 Random Forest:

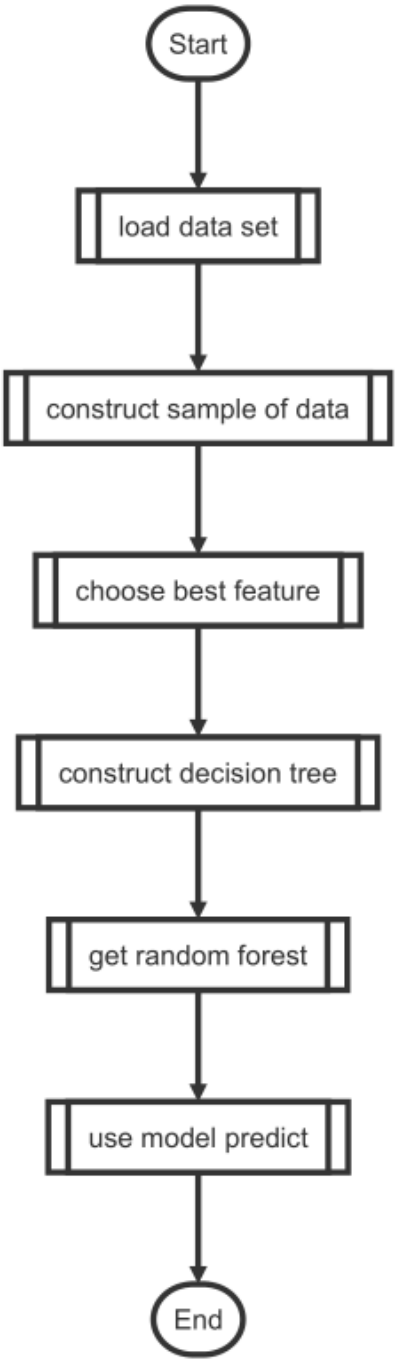
Random forest is an extension of bagging. On the basis of bagging, random attribute (*i.e.* feature) selection is used in the training process of decision tree. The traditional decision tree selects an optimal attribute from all attributes. In random forest, for each node of decision tree, a subset containing K attributes is randomly selected from the attribute set of the node, then select the optimal attribute from this subset for partition. The parameter K controls the randomness of random forest: if K is equal to the size of attributes, random forest is the same as traditional decision tree; if K is equal to 1, we will only choose one attribute to split.

In short, random forest is the combination of decision trees, but there are two differences between them:

First, sampling difference. We need to get the sample set with samples for training (m is a hyper-parameter). This can ensure the training samples of each decision tree are not exactly the same.

Second, feature selection. Features used to split of each decision tree are randomly selected from all the features. Also, the size of features is a hyper-parameter.

The procedure of random forest is as follows:



---

2. TF-IDF:

`TF-IDF` is a statistical method to evaluate the importance of a word to a file set or one of the files in a corpus. The importance of a word increases with the frequency of its appearance in the document, but decreases with the frequency of its appearance in the corpus.

The more times a word appears in an article, the less it appears in all documents, the more it can represent the article. We have the following formulas:

$$TF-IDF(t,d,D) = TF(t,d) * IDF(t,D)$$
$$TF(t,d) = log(1 + freq(t,d))$$
$$IDF(t,D) = log(\frac{N}{count(d \in D : t \in d)})$$

From above formulas, we can know that the high frequency of words in a specific file and the low frequency of words in the whole file set can produce high weight `TF-IDF`. Therefore, `TF-IDF` tends to filter out common words and retain important words.

三、Experiment Content

1. Execution Environment:

OS: `Windows 10`.

Python Version: `Python 3.6`.

Packages: `numpy`, `pandas`, `re`, `csv`, `sklearn`.

Attention: All above packages are used to process the data. `Random Forest` is implemented by myself without any packages related machine learning.

2. Experimental Steps:

First, we need to process the given data to obtain word vector, then we use `Random Forest` to finish classification according to word vector features.

```
if __name__ == '__main__':
    data = pd.read_csv('mbti_1.csv')  # read data
    process(data)  # preprocess the data
    mbti, info = read()  # read the data after preprocessing
    TFIDF(mbti, info)  # NLP
    for j in range(16):
        dataset = loaddata()  # load dataset
        dataset = tobinary(dataset, j)   # for every label(16 in all), do a
binary classification problem.
        traindata, testdata = split_train_test(dataset, traintestratio)  # get
train set and test set
        RF(traindata)                # Random Forest Classifier
        acc = getacc(testdata[:-1])  # get accuracy
        print("the accuracy of {} is:{}%".format(label(j), acc))
```

Here I follow the above steps to introduce the code structure in details:

2.1 Data Preprocessing:

By observing the data, we can see that: each row is a sample, the first column is their personality type, and the second column is their posts text content, which has multiple pieces, separated by |||.

In data preprocessing, following points to pay attention:

First, we need to encode the personality type, the method I choose is to code the overall 16 personality types with the numbers 0-15, a number represents a personality type.

Second, there are many links in the data, they can't give us useful information, so we need to remove them.

Then, the string in posts related to personality types, such as 'intp', which we need to remove because it may prevent the classifier from learning nothing.

The code of this stage is as follows:

```
def process(data):
    # encoding the personality type:
    encode = {'INTJ': 0, 'INTP': 1, 'INFJ': 2, 'INFP': 3, 'ISTJ': 4, 'ISTP': 5,
'ISFJ': 6, 'ISFP': 7, 'ENTJ': 8, 'ENTP': 9, 'ENFJ': 10, 'ENFP': 11, 'ESTJ': 12,
'ESTP': 13, 'ESFJ': 14, 'ESFP': 15}
    typeinposts = ['INFJ', 'ENTP', 'INTP', 'INTJ', 'ENTJ', 'ENFJ', 'INFP',
'ENFP', 'ISFP', 'ISTP', 'ISFJ', 'ISTJ', 'ESTP', 'ESFP', 'ESTJ', 'ESFJ', 'infj',
'entp', 'intp', 'intj', 'entj', 'enfj', 'infp', 'enfp', 'isfp', 'istp', 'isfj',
'istj', 'estp', 'esfp', 'estj', 'esfj']
    persontype = []   # store personality type
    postsinfo = []    # store posts
    for row in data.iterrows():
        # remove useless information
        posts = row[1].posts
        temp = re.sub('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|(?:%[0-9a-fA-F][0-
9a-fA-F]))+', ' ', posts)
        temp = re.sub("[^a-zA-Z]", " ", temp)
        temp = re.sub(' +', ' ', temp).lower()
        # remove the words in posts which are related to personality type,
prevent the classifier from learning nothing.
        for j in range(len(typeinposts)):
            temp = temp.replace(typeinposts[j], "")
        # get personality and posts after preprocessing:
        label = encode[row[1].type]
        persontype.append(label)
        postsinfo.append(temp)
    # write them into a new file:
    with open("pre.csv", "w", newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(["type", "posts"])
        for k in range(len(postsinfo)):
            writer.writerow([persontype[k], postsinfo[k]])

def read():
    # read the data after preprocessing, this function can reduce the call of
process(), so it save time.
    data = pd.read_csv("pre.csv")
    perlist = data['type']
    postlist = data['posts']
    perlist = np.array(perlist)
    postlist = np.array(postlist)
    return perlist, postlist

# load dataset, the last column is label, others are features.
def loaddata():
    dataset = list()
    with open('after.csv', 'r') as file:
        csv_reader = csv.reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    for i in range(len(dataset)):
        for j in range(len(dataset[0])):
            dataset[i][j] = float(dataset[i][j])
```

```
    return dataset
```

2.2 Nature Language Processing:

In this step, we get the `tfidf-matrix` which will become the information need to be trained and learned. Then we write the matrix and tags into `"after.csv"`. The code of this stage is as follows:

```
def TFIDF(perlist, postlist):
    # according to experiment and after several attempts to determine the
parameter:
    word2vec = CountVectorizer(analyzer="word", max_features=2000, max_df=0.8,
min_df=0.05)
    # calculate the frequency of words
    print("WAIT...")
    freq = word2vec.fit_transform(postlist)
    # print(freq)
    # calculate tf-idf matrix:
    tftrans = TfidfTransformer()
    tfidf = tftrans.fit_transform(freq).toarray()
    # print tf-idf matrix:
    # print(tfidf, len(tfidf),len(tfidf[0]))
    tfidf = np.mat(tfidf)
    finals = np.c_[tfidf, perlist]
    # print(finals.shape[0], finals.shape[1])
    # save input features(tf-idf matrix) and output label in 'after.csv'(the last
column is label)
    np.savetxt('after.csv', finals, delimiter=',')
```

2.3 Random Forest:

I use `Random Forest` implemented by myself to classify. We need to mark the label of the personality type which to be predicted in this round as `1`, the other 15 types as `0` at the beginning of classification. The core code is as follows:

```
def tobinary(data, K):
    # (data, K)==(dataset, the label of positive samples)
    # transfer multi-class into binary-class, convert the label of positive
samples into 1, others into 0:
    for i in range(len(data)):
        if data[i][-1] == K:
            data[i][-1] = 1
        else:
            data[i][-1] = 0
    return data

if __name__=="__main__":
    for j in range(16):
        dataset = loaddata()
        dataset = tobinary(dataset, j)   # for every label(16 in all), do a
binary classification problem.
        traindata, testdata = split_train_test(dataset, traintestratio)  # get
train set and test set
        # myRF = randomForest(numoftrees, maxdepth, minsize, sample_ratio,
feature_ratio)
```

```
        RF(traindata)                # Random Forest Classifier
        acc = getacc(testdata[:-1])  # get accuracy
        print("the accuracy of {} is:{}%".format(label(j), acc))
```

Then I will introduce implementation procedure of `Random Forest`:

First, divide the dataset into train set and test set randomly. The ratio of training set and test set is 70% and 30%.

```
# train set:test set=7:3(ratio=0.3)
def split_train_test(dataset, ratio):
    num = len(dataset)
    train_num = int((1-ratio) * num)
    dataset2 = list(dataset)
    traindata = list()
    while len(traindata) < train_num:
        index = randint(0, len(dataset2)-1)    # random choose train set
        traindata.append(dataset2.pop(index))
    testdata = dataset2                        # dataset pop train set, remains
are test set.
    return traindata, testdata
```

Then, we need to choose sample randomly. The size of sample is as large as the square root of dataset. We need to do a sampling with put back which ensures that the training samples of each decision tree are not exactly the same.

```
# choose sample random, the size of sample is the square root of data in all.
def choosesample(dataset):
    sample = []
    n_sample = 2 * np.sqrt(len(dataset))
    while len(sample) < n_sample:
        index = randint(0, len(dataset) - 2)
        sample.append(dataset[index])
    return sample
```

Then we need to choose the best feature to split for each decision tree. How to choose best feature? I use Gini index, information gain and gain ratio as evaluation criteria. For Gini index, the feature with the smallest value is the best. For the other two, the feature with the largest value is the best.

However, there is a problem: the attribute values of the given data are continuous. According to the *watermelon book* (*Zhihua Zhou*), we need to calculate the median after sorting, then split the data by comparing the attribute value with the median. We need to calculate the above evaluation criteria, the most appropriate one is selected for splitting. Here, I directly select a row's attribute value instead of calculating the median for convenience, then select the best one to split. The code of this part:

```
# get the best feature to split(use Gini, gain ratio and information gain three
criteria)
def bestsplit(dataset, n_features):
    class_values = list(set(row[-1] for row in dataset))  # sort
    b_index, b_value, b_score, b_groups = 9999, 9999, 9999, None
```

```
    features = list()
    while len(features) < n_features:
        index = randint(0, len(dataset[0]) - 2)  # choose features random
        if index not in features:
            features.append(index)
    for index in features:
        for row in dataset:             # use values in this feature of each
sample to split
            groups = splitofeachtree(index, row[index], dataset)  # divide data
into two group
            # as said before(in function define), for these three criteria, the
smaller value, the better effect
            gini = GINI(groups, class_values)         # calculate criteria to
judge the best feature for splitting
            # entofleaf = EntofLeaf(groups, class_values)
            # gain_ratio = gainratio(groups, class_values)
            # if gain_ratio < b_score:
            #     b_index, b_value, b_score, b_groups = index, row[index],
gain_ratio, groups
            # if entofleaf < b_score:
            #     b_index, b_value, b_score, b_groups = index, row[index],
entofleaf, groups
            if gini < b_score:
                b_index, b_value, b_score, b_groups = index, row[index], gini,
groups
    return {'index': b_index, 'value': b_value, 'groups': b_groups}

# Gini index, the smaller value, the better effect
def GINI(groups, class_values):
    gini = 0.0
    total_size = 0
    for group in groups:
        total_size += len(group)
    for group in groups:
        size = len(group)
        if size == 0:
            continue
        for class_value in class_values:
            proportion = [row[-1] for row in group].count(class_value) /
float(size)
            gini += (size / float(total_size)) * (proportion * (1.0 -
proportion))
    return gini

# this function calculate the entropy of leaf node
# due to the entropy of root is a constant(for the same feature)
# we know that the bigger Information Gain, the better effect.
# so the smaller value of this function return, the better effect.
def EntofLeaf(groups, class_values):
    total_size = 0
    leafent = 0.0
    for group in groups:
        total_size += len(group)
    for group in groups:
        temp = 0.0
        size = len(group)
        if size == 0:
            continue
```

```
        for class_value in class_values:
            proportion = [row[-1] for row in group].count(class_value) /
float(size)
            temp += (plogp(proportion))
        leafent += (temp * (size / float(total_size)))
    return -leafent

# calculate the opposite number of gain ratio, so the smaller value, the better
effect
def gainratio(groups, class_values):
    entofleaf = EntofLeaf(groups, class_values)      # get the entropy of leaf
which by splitting
    total_size = 0
    ent = 0.0
    for group in groups:
        total_size += len(group)
    for class_value in class_values:
        count = 0
        for group in groups:
            count += [row[-1] for row in group].count(class_value)
        prob = count / float(total_size)
        ent += (plogp(prob))
    ent = -ent                # get the entropy of root
    InfoGain = ent - entofleaf      # calculate the information gain
    entofclass = 0.0
    for group in groups:
        size = len(group)
        prod = size / float(total_size)
        entofclass += (plogp(prod))       # get the opposite number of IV(a)
(the denominator of gain ratio)
    ratio = 999.0 if entofclass == 0.0 else (InfoGain / entofclass)    # get the
opposite number of gain ratio
    return ratio
```

After we get the best feature, we can split the data according to the value of this feature and construct each decision tree.

```
# split of every decision tree
def splitofeachtree(index, value, dataset):
    left = []
    right = []
    # left, right = list(), list()
    for row in dataset:
        if row[index] < value:
            left.append(row)
        else:
            right.append(row)
    return left, right

def build_one_tree(train, maxdepth, minsize, n_features):
    root = bestsplit(train, n_features)
    subclassifier(root, maxdepth, minsize, n_features, 1)
    return root
```

Finally, we combine these decision trees to get the random forest:

```
# Random Forest Algorithm
def RF(train):
    n_features = np.sqrt(len(train[0]))  # the size of features used to split ==
the square root of features in all
    for i in range(numoftrees):
        sample = choosesample(train)
        tree = build_one_tree(sample, maxdepth, minsize, n_features)
        trees.append(tree)
    return trees
```

At this point, we can use this model to calculate the accuracy on test set:

```
# calculate accuracy in test set
def getacc(testdata):
    correct = 0
    for i in range(len(testdata)):
        predicted = bagging(testdata[i])
        if testdata[i][-1] == predicted:
            correct += 1
    return correct / float(len(testdata)) * 100.0

# vote of random forest(bagging)
def bagging(onetestdata):
    predictions = [predict(tree, onetestdata) for tree in trees]
    return max(set(predictions), key=predictions.count)

# predict with recursion in a decision tree
def predict(node, row):
    if row[node['index']] < node['value']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']
```

---

3. Experimental Result

3.1 Hyper-Parameter Choosing:

In random forest, these parameters we need to adjust by manual:

`numoftrees` : the number of decision tree in random forest;

`maxdepth` : the max depth of each tree;

`minsize` : the min size of each leaf node;

`s-samples` : the number of samples selected by random sampling;

`s-features` : the number of features selected randomly.

Obviously, `minsize`=1 is appropriate. The parameter `numoftrees` is less important than others. So I assign the value 20 to it by a few attempts. For `s-samples` and `s-features` , we always choose $\sqrt{sizeofdata}$ and $\sqrt{features}$ from experiment. So we only need adjust `maxdepth` by manual.

For the parameter `maxdepth`, I use Gini index test the accuracy of 'INTP' for different value. As we all know, if the tree is too deep, it will reduce the searching efficiency, also may produce over-fitting. If the depth is too small, it will affect the training effect. Finally, I think the best value of `maxdepth` is 10.

So I assign the following values to these parameters:

$$
\begin{aligned}
numoftrees &= 20 \\
maxdepth &= 10 \\
minsize &= 1 \\
s-samples &= \sqrt{sizeofdata} \\
s-features &= \sqrt{features}
\end{aligned}
$$

3.2 Predict On Test Set:

The accuracy of this model in the test set are listed in the table. Due to the randomness of algorithm, the results have difference each time, but the difference is very small which less than 1%. The result in the table is one of them:

	Gini index	Information gain	Gain ratio
INTJ	88.2014%	87.1637%	89.0469%
INTP	84.0507%	84.0123%	84.0507%
INFJ	83.2052%	82.3981%	85.0500%
INFP	79.0930%	78.4781%	78.0169%
ISTJ	97.4635%	97.5788%	97.8094%
ISTP	95.8493%	95.7340%	95.8493%
ISFJ	97.8478%	98.0400%	98.0015%
ISFP	96.6948%	97.2329%	96.9639%
ENTJ	97.4635%	97.3482%	97.1945%
ENTP	92.3905%	91.8524%	92.3520%
ENFJ	97.7325%	97.4635%	97.7709%
ENFP	92.4289%	91.8140%	92.4289%
ESTJ	99.5003%	99.4235%	99.6925%
ESTP	99.1161%	99.0007%	99.0776%
ESFJ	99.5772%	99.5388%	99.5388%
ESFP	99.3851%	99.3082%	99.2698%

From the table and the table of the last experiment (`SVM`), we can get the conclusion that random forest has higher accuracy than `SVM`, that is to say, random forest classifier has better effect than `SVM` in this classification problem. Also, we can get the best effect when we use gain ratio as evaluation criteria. By counting the number of data in each class, I found that the data in the latter classes is very small, so the accuracy of these classes is so high.