# Machine Learning Term Project Report

109550186 – 李嘉玲

## I. Preprocessing Procedure

- Missing value in numerical features (solved by data imputation)

```
# df1n2.columns[df1n2.isnull().any()]
df1n2.isnull().sum()
```

```
fixed_acidity           65
volatile_acidity        44
citric_acid             60
residual_sugar          39
chlorides               52
free_sulfur_dioxide     57
total_sulfur_dioxide    45
density                 46
pH                      58
sulphates               51
alcohol                 58
class                    0
dtype: int64
```

```
[652] df1n2 = df1n2.fillna(df1n2.mean())
```

```
[653] df1n2.isnull().sum()
```

```
fixed_acidity           0
volatile_acidity        0
citric_acid             0
residual_sugar          0
chlorides               0
free_sulfur_dioxide     0
total_sulfur_dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
class                   0
dtype: int64
```

The method used is filling the missing values with mean values of the non-empty value from that feature. In the code above, it can be seen that I used the panda's library *dataframe.fillna()* to replace the NaN value in the dataset.

- Detecting outliers (includes data cleaning and reduction)

```
[654] len(df.loc[outliers(df,df.columns[:-1])]) # number of outliers
```

```
0
```

```
[655] df = df.drop(outliers(df,df.columns[:-1]),axis = 0).reset_index(drop = True)
```

First, I tried to output the number of outliers to see if there is outlier in the dataset. If there is outlier, it will be dropped from the dataset and index will be reset. Again, I used the panda's library to detect and solve this issue.

- For optimized result (data transformation)

```
[663] # get optimized result
      sc = StandardScaler()

      X_train = sc.fit_transform(X_train)
      X_val = sc.fit_transform(X_val)
```

The standard scaler from sklearn preprocessing library is used to removes mean and scales the variables to unit variance.

- Preprocessing data for caategorical values

```
[137] # Preprocess data before training
    from keras.preprocessing.text import text_to_word_sequence
    def preprocess_data(df):
        reviews = []
        for raw in tqdm(df['Phrase']):
            # print(raw)
            text = raw.replace("'","")
            tokenized_train_data = text_to_word_sequence(text,filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n',split=" ")
            stop_words = set(stopwords.words('english'))
            removesw = [i for i in tokenized_train_data if not i in stop_words]
            rswtext = ' '.join(removesw)
            numberRemove = ''.join(num for num in rswtext if not num.isdigit())
            stemmer = PorterStemmer()
            stem_input = nltk.word_tokenize(numberRemove)
            stem_text = ' '.join([stemmer.stem(word) for word in stem_input])
            reviews.append(stem_text)
        return reviews
```

For categorical values, the methods used are removing special characters like parantheses and other symbols and stop words, as well as morphological and inflexional endings from the sentences given in the variable phrase of the dataframe.

## II. Decision Tree Classifier Algorithm Snapshots

Decision tree classifier built using the gini and entropy measurement as shown in the functions below. Function *gini* determines the counts of each distinct label in the set. The likelihood of each label is then calculated, and one minus the sum of the squares of these probabilities is returned. If the labels are pure, then the Gini impurity will be low. While function *entropy* determines the counts of each distinct label in the set. The chance of each label is then calculated, and the result is the negative sum of the probabilities multiplied by the probabilities logarithm.

```python
# functions used to construct the decision tree classifier
def gini(sequence, weights=None):
    if weights is None: # count 1 - sum of square of probabilities
        _, counts = np.unique(sequence, return_counts=True)
        p = (counts / len(sequence)) ** 2
        return 1.0 - np.sum(p)
    else:
        tot = 0
        weights = weights / weights.sum()
        for c in np.unique(sequence):
            #change prob become weighted prob
            tot = np.sum(weights[sequence == c]) ** 2
        return 1 - tot


def entropy(sequence, weights=None):
    if weights is None:
        _, counts = np.unique(sequence, return_counts=True)
        p = counts / len(sequence)  #The Probability
        return -np.sum(p * np.log2(p))
    else:
        entropy = 0
        weights = weights / weights.sum()
        for c in np.unique(sequence):
            # calculate the weighted probability
            tmp = np.sum(weights[sequence == c])
            entropy -= tmp * np.log2(tmp)
        return entropy
```

The image below shows several variables of this decision tree, which are criterion, max_depth, max_features and n_features. Criterion is used to measure the quality of a split, with default Gini impurity, but also can be entropy. Max_depth is the maximum depth the tree can have. Max_features and n_features respectively shows the most features the tree will take into account while determining the appropriate split at each node and the number of features in the input data.

```python
def __init__(self, criterion='gini', max_depth=None, max_features=None):
    self.criterion = globals()[criterion]

    self.max_depth = max_depth if max_depth is not None else 2 ** 100
    self.max_features = max_features
    self.n_features = None
```

Function *fit* is the one that build the decision tree. It takes input of training data and target labels as well as optional sample weight. While *get_node* function is called recursively by *fit* to build the decision tree. It takes input of input data, target labels, current node depth, and optional sample weight. Then, it will return Node, which if current depth greater than or equal to max_depth, will return a leaf node with prediction equal to most common class in data. Else, it will find best feature and threshold to split the data at this node by calling *best_split*, and it recursively calls *get_node* on the data in the left and right child nodes.

```
def fit(self, x_data, y_data, sample_weight=None):
    self.n_features = x_data.shape[1]
    self.root = self.get_node(x_data, y_data, depth=0, sample_weight=sample_weight)

def get_node(self, x, y, depth, sample_weight=None):
    weighted_counts = np.bincount(y, weights=sample_weight)
    prediction = np.argmax(weighted_counts)

    node = Node(c_value=self.criterion(y, sample_weight), prediction=prediction)
    if depth >= self.max_depth:
        return node

    node.feature_idx, node.threshold = self.best_split(x, y, sample_weight)
    if node.feature_idx is None:
        return node

    left_idx = x[:, node.feature_idx] < node.threshold
    x_left, y_left = x[left_idx], y[left_idx]
    x_right, y_right = x[~left_idx], y[~left_idx]

    # get child nodes recursively
    if sample_weight is not None:
        node.left = self.get_node(x_left, y_left, depth=depth + 1, sample_weight=sample_weight[left_idx])
        node.right = self.get_node(x_right, y_right, depth=depth + 1, sample_weight=sample_weight[~left_idx])
    else:
        node.left = self.get_node(x_left, y_left, depth=depth + 1)
        node.right = self.get_node(x_right, y_right, depth=depth + 1)

    return node
```

Function *best_function* is used to find best feature and threshold to split data. If max_features is set, it randomly selects a subset of the features to consider when searching for the best split. *Predict* function returns a list of predicted labels by traversing tree and returning predicition at the leaf node that the sample ends up.

```
def best_split(self, x, y, sample_weight):
    if len(y) <= 1:
        return None, None

    parent_c = self.criterion(y, sample_weight)
    best_infog = -2 ** 64
    best_idx, best_th = None, None

    if self.max_features is not None:
        available_features = np.random.choice(np.arange(self.n_features), size=self.max_features, replace=False)
    else:
        available_features = np.arange(self.n_features)

    for idx in available_features:
        sort_idx = np.argsort(x[:, idx])
        thresholds = x[sort_idx, idx]
        labels = y[sort_idx]

        for pos in range(1, len(y)):
            if thresholds[pos] == thresholds[pos - 1]:
                continue

            if sample_weight is not None:
                sorted_sample_weight = sample_weight[sort_idx]
                left_c = self.criterion(
                    labels[:pos], sorted_sample_weight[:pos])
                right_c = self.criterion(
                    labels[pos:], sorted_sample_weight[pos:])
            else:
                left_c = self.criterion(labels[:pos])
                right_c = self.criterion(labels[pos:])
                  else:
                      left_c = self.criterion(labels[:pos])
                      right_c = self.criterion(labels[pos:])

                  child_c = (pos * left_c + (len(y) - pos) * right_c) / len(y)
                  infog = parent_c - child_c

                  if infog > best_infog:
                      best_infog = infog
                      best_idx = idx
                      best_th = (thresholds[pos] + thresholds[pos - 1]) / 2

          return best_idx, best_th

      def predict(self, x_data):
          def util(self, x):
              cur_node = self.root
              while cur_node.left and cur_node.right:
                  if x[cur_node.feature_idx] < cur_node.threshold:
                      cur_node = cur_node.left
                  else:
                      cur_node = cur_node.right

              return cur_node.prediction
          return np.stack([util(self, single_x) for single_x in x_data])
```

Snippets of input and output in dataset 1 and 2:

```
[ ] # Read the csv files inserted
    df = read_file('./X_train.csv')
    df2 = read_file('./y_train.csv')
    df3 = read_file('./X_test.csv')
```

```
df.head()
```

| | fixed_acidity | volatile_acidity | citric_acid | residual_sugar | chlorides | free_sulfur_dioxide | total_sulfur_dioxide | density | pH | sulphates | alcohol |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 7.0 | 0.23 | 0.40 | 1.6 | 0.063 | NaN | 67.0 | 0.99520 | 3.50 | 0.63 | 11.1 |
| **1** | 7.8 | 0.60 | 0.26 | 2.0 | 0.080 | 31.0 | 131.0 | 0.99622 | NaN | 0.52 | 9.9 |
| **2** | 9.7 | 0.69 | 0.32 | 2.5 | 0.088 | 22.0 | 91.0 | 0.99790 | 3.29 | 0.62 | 10.1 |
| **3** | 12.0 | 0.38 | 0.56 | 2.1 | 0.093 | 6.0 | 24.0 | 0.99925 | 3.14 | 0.71 | 10.9 |
| **4** | 6.4 | 0.64 | 0.21 | 1.8 | 0.081 | 14.0 | 31.0 | 0.99689 | 3.59 | 0.66 | NaN |

```
[ ] df.shape
```

```
(1023, 11)
```

```
[ ] df.describe()
```

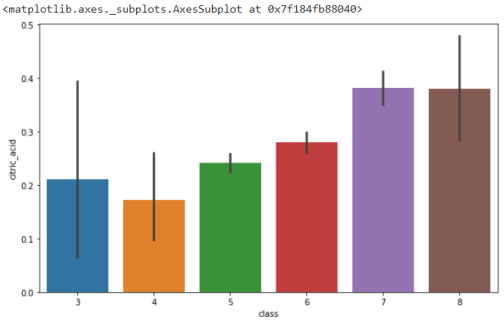
| | fixed_acidity | volatile_acidity | citric_acid | residual_sugar | chlorides | free_sulfur_dioxide | total_sulfur_dioxide | density | pH | sulphates | alcohol |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 958.000000 | 979.000000 | 963.000000 | 984.000000 | 971.000000 | 966.000000 | 978.000000 | 977.000000 | 965.000000 | 972.000000 | 965.000000 |
| **mean** | 8.373800 | 0.526118 | 0.274216 | 2.510010 | 0.087151 | 15.920807 | 45.801125 | 0.996776 | 3.308632 | 0.663580 | 10.445009 |
| **std** | 1.776455 | 0.176769 | 0.193308 | 1.270972 | 0.043777 | 10.158979 | 33.314198 | 0.001894 | 0.153336 | 0.176466 | 1.048401 |
| **min** | 4.600000 | 0.120000 | 0.000000 | 1.200000 | 0.012000 | 1.000000 | 6.000000 | 0.990200 | 2.740000 | 0.370000 | 8.400000 |
| **25%** | 7.100000 | 0.390000 | 0.100000 | 1.900000 | 0.071000 | 8.000000 | 22.000000 | 0.995650 | 3.210000 | 0.550000 | 9.550000 |

```
[ ] df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1023 entries, 0 to 1022
Data columns (total 11 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed_acidity         958 non-null    float64
 1   volatile_acidity      979 non-null    float64
 2   citric_acid           963 non-null    float64
 3   residual_sugar        984 non-null    float64
 4   chlorides             971 non-null    float64
 5   free_sulfur_dioxide   966 non-null    float64
 6   total_sulfur_dioxide  978 non-null    float64
 7   density               977 non-null    float64
 8   pH                    965 non-null    float64
 9   sulphates             972 non-null    float64
 10  alcohol               965 non-null    float64
dtypes: float64(11)
memory usage: 88.0 KB
```

```
# higher citric acid, higher quality
fig = plt.figure(figsize = (10,6))
sns.barplot(x = 'class', y = 'citric_acid', data = df1n2)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f184fb88040>
```




Barplot features show that some feature might not be that crucial and can have less weight compared to features with higher role in the prediction algorithm.

## III. Result

Dataset 1 training dataset accuracy:

```
# predict validation set
y_pred = clf.predict(X_val)
report(y_pred, y_val)
```

```
Accuracy: 0.6
Confusion Matrix:
[[ 0  4  3  0  0]
 [ 0 62 18  1  0]
 [ 0 30 45 11  0]
 [ 0  3  8 16  0]
 [ 0  1  1  2  0]]
Classification Report:
              precision    recall  f1-score   support

           4       0.00      0.00      0.00         7
           5       0.62      0.77      0.69        81
           6       0.60      0.52      0.56        86
           7       0.53      0.59      0.56        27
           8       0.00      0.00      0.00         4

    accuracy                           0.60       205
   macro avg       0.35      0.38      0.36       205
weighted avg       0.57      0.60      0.58       205
```

Dataset 2 training dataset accuracy:

```
# predict validation set
y_pred = clf.predict(X_val_)
report(y_pred, y_val)
```

```
Accuracy: 0.525830997196636
Confusion Matrix:
[[   45   106   931    39    11]
 [   42   357  3782   151    32]
 [   20   282 12118   253    60]
 [   26   130  4487   509   116]
 [    7    28  1133   204   101]]
Classification Report:
              precision    recall  f1-score   support

           0       0.32      0.04      0.07      1132
           1       0.40      0.08      0.14      4364
           2       0.54      0.95      0.69     12733
           3       0.44      0.10      0.16      5268
           4       0.32      0.07      0.11      1473

    accuracy                           0.53     24970
   macro avg       0.40      0.25      0.23     24970
weighted avg       0.47      0.53      0.42     24970
```

# IV. Submission

- Preprocessed training dataset: X_train_final
- Preprocessed testing dataset: X_test_final
- Predicted result of testing dataset: y_test_final