For this report, I upload two files. One is .ipynb file which contains the process of researching with the markdown explanation. The other is the code for you to reproduce the result showed below (Due to the training data split, you might not be able to reproduce the exact performance).

Develop environment

Language: python3.8

UI snapshot

Just follow the sequence in the jupyter notebook to run.

Dataset 1

Data preprocessing

There are three preprocessing issue I concern. 1. Missing value 2. Outlier 3. sampling.

Missing value

First, I consider to use the simplest way to fill the missing value. To look into the data, there are the skewed distribution in some features (following figure), so we better choose median imputation instead of mean imputation.

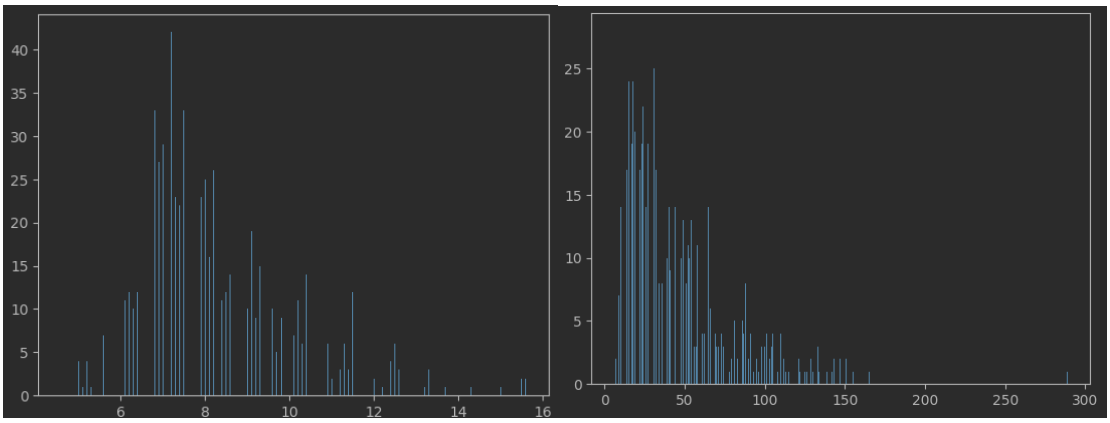

Figure of distribution of features: fixed acidity and total_sulfur_dioxide.

Next, I find the total number of missing value in this dataset is 575, which is about 15% of whole dataset. According to the paper[1], it's reasonably safe to use the data imputation, so I try to use kNN model to predict the missing value. I find all rows that don't contain the missing value in any column. Next, I calculate the mean and standard error of to check if the missing is random.

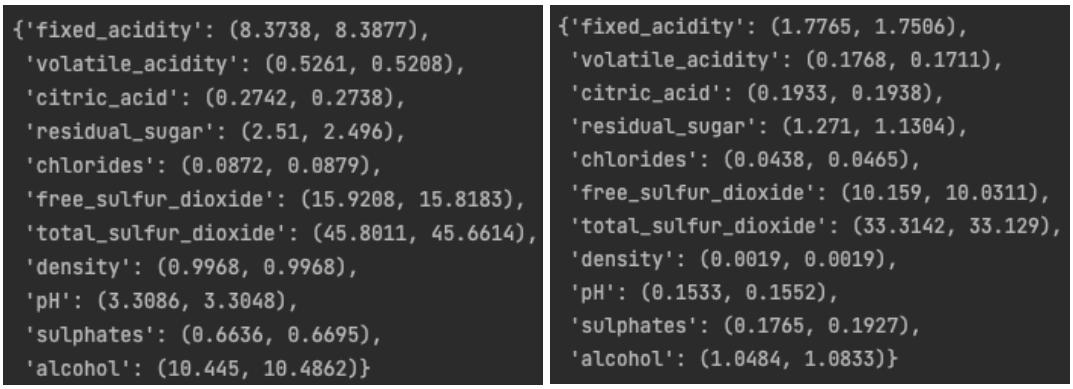

Figure of comparison of mean and std between original dataset and dataset without missing values.

It seems that the missing value is distributed randomly, so we can use kNN to predict the missing value. The algorithm is detailed following:

1. find all rows that don't contain the missing value in any column
2. rescale all column to avoid a column being dominate
3. calculate the distance of other non-missing features using the Euclidean distance.
4. Calculate mean of the k nearest instances as the imputation data.
   I use the function of **`minmax_scale`** from **`sklearn`** to rescale the dataframe easily and quickly because they have high compatibility. The concept of rescaling is quite easy to implement. However, to show I understand the concept of rescaling, I will explain it here.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

   X_norm is the number after rescaling.

The result seems reasonable because when I go back to check the nearest instances, I find their variance is quite low.

Outlier

After filling up the missing value, we can now try to remove the outlier. I calculate the zscore of each row in terms of each column and remove the rows which contain the outlier (value>3) in at least one column. After removing, the number of instances drop from 1023 to 926, which is about 9%.

Sampling

After overviewing the data, we can see the dataset is very inbalance.

	target
5	427
6	417
7	124
4	33
8	14
3	8

	target
5	385
6	384
7	110
4	30
8	11
3	6

Figure of the number of target features before and after outlier removing.
There are three candidate method to deal with it. 1. over-sampling 2. under-sampling 3. Sample weight. First, I don't take the under-sampling into consideration because the minimum number of features is too low (only 6), which lose too many data and might lead to a model with a huge bias. Thus, the random over-sampling will be my first shot. However, the difference between the maximum and minimum is quite large, which might lead to a bias, too. Last, I will try to use the sample weight, which will give more punishment when the tree misclassify the class with small volume. The detail will be introduced in the tree building part.

Building Tree

The basic structure using for building a tree is to create a root node, and then find the best features with corresponding value threshold to split the data into 2 groups. The method used to find the best feature and threshold is to iterate all the features and the values and use the middle point of value and the smallest larger value as the threshold to calculate IG. After find the most optimal feature and value, we store in this node and split data into two group to create two subtree for the node. We must choose the algorithm to split the data and decide what target will be stored the leaves (the last node in each path).

First Version

In this version, I build the tree with two key designs. 1. Pre-pruning 2. Information gain

Pre-pruning

I set two parameter to avoid the tree grow too large and deep. 1. Max_depth 2. Min_samples_at_leaf. The first one is to stop the tree growing when it reach the specific depth. The second is to stop the tree growing when the instance in the node is too small.

Information gain

I use the information gain instead of entropy to select the best feature and threshold when the tree is splatted. I iterate through all features with the value of

instance to find which one can have the best information gain.

Before training, I use the **`RandomOverSampler`** from **`imblean`** to do oversampling. Next I use **`train_test_split`** from **`sklearn`** to split the data into train and value with ration of 20%.

Following is the result based on the dataset with random over-sampling.

The column of {0,1,2,3,4,5} is corresponding to the class of {3,4,5,6,7,8}.

```
Accuracy: 0.8246753246753247
Confusion Matrix:
[[77  0  0  0  0  0]
 [ 0 77  0  0  0  0]
 [ 2  6 53 11  5  0]
 [ 0  2 37 26  9  3]
 [ 0  0  3  2 71  1]
 [ 0  0  0  0  0 77]]
```

```
Classification Report:
              precision    recall  f1-score   support

           3       0.97      1.00      0.99        77
           4       0.91      1.00      0.95        77
           5       0.57      0.69      0.62        77
           6       0.67      0.34      0.45        77
           7       0.84      0.92      0.88        77
           8       0.95      1.00      0.97        77

    accuracy                           0.82       462
   macro avg       0.82      0.82      0.81       462
weighted avg       0.82      0.82      0.81       462
```

We can see that although the accuracy and the average precision, recall and f1-score is highe, the recall and precision of target 5 and 6 are very low for the. There are a huge gape between these two and others. It is because we oversample the target with replacement, so the model can distinguish the instances the oversampling instances easily(their description feature are all the same).Thus, we can infer that this model has severe bias . To avoid this bias, we should not use the oversampling. However, following are the result without using oversampling. It is easy to see that the performance is incredibly bad. The next version, the aim is to improve the overall performance without bias.

```
Accuracy: 0.5698924731182796
Confusion Matrix:
[[ 0  0  1  0  0  0]
 [ 0  0  4  2  0  0]
 [ 0  1 45 32  0  0]
 [ 0  4 16 52  5  0]
 [ 0  0  6  7  9  0]
 [ 0  0  2  0  0  0]]
```

```
Classification Report:
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       0.00      0.00      0.00         6
           5       0.61      0.58      0.59        78
           6       0.56      0.68      0.61        77
           7       0.64      0.41      0.50        22
           8       0.00      0.00      0.00         2

    accuracy                           0.57       186
   macro avg       0.30      0.28      0.28       186
weighted avg       0.56      0.57      0.56       186
```

Second Version

In this version, I add the boosting technique to the tree to deal with the issue of imbalance data. I reference to the Adaboost.m1 algorithm, which is the modified version of Adaboost for the multi-class classification task. To do so, we need to implement an extra function called boost and an extra list called weight.

Input: Training set $S = \{\mathbf{x}_i, y_i\}$, $i = 1, \ldots, N$; and $y_i \in \mathbb{C}, \mathbb{C} = \{c_1, \ldots, c_m\}$; T: Number of iterations; I: Weak learner
Output: Boosted classifier:

$$H(x) = \arg\max_{y \in \mathbb{C}} \sum_{t=1}^{T} \ln\left(\frac{1}{\beta_t}\right) [h_t(x) = y] \text{ where } h_t, \beta_t$$

are the induced classifiers (with $h_t(x) \in \mathbb{C}$) and their assigned weights, respectively

$D_1(i) \leftarrow 1/N$ for $i = 1, \ldots, N$
for $t = 1$ to T **do**
  $h_t \leftarrow I(S, D_t)$
  $\varepsilon_t \leftarrow \sum_{i=1}^{N} D_t(i)[h_t(\mathbf{x}_i) \neq y_i]$
  if $\varepsilon_t > 0.5$ **then**
    $T \leftarrow t - 1$
    return
  end if
  $\beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$
  $D_{t+1}(i) = D_t(i) \cdot \beta_t^{1-[h_t(\mathbf{x}_i) \neq y_i]}$ for $i = 1, \ldots, N$
  Normalize D_{t+1} to be a proper distribution
end for

The figure of the algorithm of Adaboost.m1.

The boost function is to reproduce this algorithm. And the weight list is the D in the figure which represent the concept of sample weight. We can use the first version tree as the weak learner. There are a key concept which didn't shown in the figure. That is, how to use the weight to build the trees, or say, calculate the information gain?

To calculate the information gain with sample weight, we should modify the P in the formula of entropy

$$Entropy(D) = -\sum_{i=1}^{c} \mathrm{P}_i * log_2 P_i$$

The original P is calculated by

$$the\ number\ of\ class\ c\ /\ the\ total\ instance\ number$$

We should modify it into

$$the\ sum\ of\ weight\ of\ class\ c\ /\ the\ total\ weight$$

After training a weak learner, we should save the value of beta because we use the $ln(1/\beta)$ as the weight of weak learner when we predict the class.

Following are the result of Adaboost.m1 decision tree without oversampling.

```
Accuracy: 0.8225806451612904
Confusion Matrix:
[[ 0  0  1  0  0  0]
 [ 0  3  3  0  0  0]
 [ 0  0 71  6  1  0]
 [ 0  0 10 62  5  0]
 [ 0  0  0  5 17  0]
 [ 0  0  0  2  0  0]]
```

```
Classification Report:
              precision    recall  f1-score   support

           3       0.00      0.00      0.00         1
           4       1.00      0.50      0.67         6
           5       0.84      0.91      0.87        78
           6       0.83      0.81      0.82        77
           7       0.74      0.77      0.76        22
           8       0.00      0.00      0.00         2

    accuracy                           0.82       186
   macro avg       0.57      0.50      0.52       186
weighted avg       0.81      0.82      0.81       186
```

We can see that the performance has a huge improvement with the same

dataset. The accuracy is raised from 0.569 to 0.822 and there are not apparent gape between targets. However, the number of target 3 and 8 is too small to correctly tell if the model can classify it or not.

Dataset 2

Preprocessing

Compared with the dataset 1, there are a lot of things we need to do for the dataset2. To make the string be used in the decision, we need to do the following work.

Normalization

First, we need to segment (tokenize) the string into a list of words and remove the word which have no meaning, for example: punctuation, article, etc. Next, we lemmatize the word which will convert the word into base. After doing these two steps, the dataset is quite "clean" right now. All these mov However, there might be some sentences will contain no word after cleaning, so we need to remove these sentences. All these above function can be performed with `preprocess_text` from `text preprocessing` except the remove the empty sentence. Thus, I modify the module to add this function.

Even though the dataset is quit clean, we still cannot use the string datatype to build the decision. If we use `tokenizer` from `keras` like the code in demo, we can index each unique word in the dataset and transfer them into the number so we can use the them in the decision tree. However, the length of the sentence in the dataset is different, so we need to use the max length of sentence as the length of our dataset and padding the sentence with 0 which are not long enough. Here, I modify the code from demo. I change the padding from pre to post because I think the sequence of word in the sentence should be consistent for the decision tree.

Building Tree

As we mentioned in the preprocessing part, the number of words represent the index of word in the dataset. That is, the relationship between the value size of the numbers is meaningless, so we can't use a threshold to split the dataset. To define the correct features, I first consider using the TFIDF (Term Frequency – Inverse document frequency) as the feature to split. To explain, following is the formula

$$Term\ frequency = \frac{number\ of\ terms(\mathrm{t})\ appears\ in\ document}{Totalnumberoftermsinadocument}$$

$$Inverse\ document\ frequency = log\frac{Total\ number\ of\ document}{Number\ of\ document\ with\ terms(t)}$$

After calculating every word score for the dataset, we can transfer every sentence in dataset as the combination of score. For example:

For three sentences:
['I am happy', 'I am sad', 'You are angry']
We can get TFIDF table:

	am	angry	are	happy	sad	you
0	1.287682	0.000000	0.000000	1.693147	0.000000	0.000000
1	1.287682	0.000000	0.000000	0.000000	1.693147	0.000000
2	0.000000	1.693147	1.693147	0.000000	0.000000	1.693147

The number of rows is the number of instances and the number of columns is the number of unique words in the dataset, that is, the dimension of dictionary. In this case, we can use the word in the columns as the feature with the corresponding value to split data.

However, our dataset is too huge so it will lead to a huge sparce matrix (114848*15235) if we calculate the TFIDF. (the ram in my computer can't even load it). Even thought we can load it into the memory, the training time will be extremely long. I try to use the bagging method to shrink the size of the dataset, but I can't make the features in columns be the same because it's a random sampling. Thus, it still not a feasible method.

I have no idea how to shrink the dimension of TFIDF score table reasonably. Besides, I can't figure out any alternative method to define the features for the dataset. At no choice, I try to use the word index or shrink the TFIDF score table brutally to build the tree. The first one will classify all instance as target 2 because it has the most number instances.

For the second one, to brutally shrink the size of TFIDF score table, I choose the top 3000 frequent words as the features. Moreover, there are another issue that is calculation time. Even though I shrink the size, for the worse case (every leaf is in the max depth) with max depth of 11, we need to iterate 114848*3000*2^11. It's too large. Thus, rather than iterate all instance to find the best threshold, I calculate the 100 equal points between max and min values as the candidate threshold to find the best threshold. I still need about 3 hours to build this model, and the performance is incredibly bad. It predict every instance as target 1. I guess it might because the

information losing. In the end, I still can't find a way to use decision tree to do sentiment analysis on a large dataset. Or even, I suspect that this is not a good option to use decision tree to do the sentiment analysis on a large dataset.

1. Madley-Dowd, P., et al., *The proportion of missing data should not be used to guide decisions on multiple imputation.* J Clin Epidemiol, 2019. **110**: p. 63-73.