ML Fianl_Project

Code

- utils function
  - calculate the entropy：follow the entropy formula

$$E(S) = \sum_{i=1}^{c} -p_i log_2 p_i$$

```
def entropy(x):
    '''
    calculates entropy of x

    input_ : x (a list of values)
    output : float, entropy value
    '''
    counts = np.bincount(np.array(x, dtype=np.int64))
    percentages = counts / len(x)

    # Caclulate entropy

    entropy = 0
    for p in percentages:
        if p > 0:
            entropy += p * np.log2(p)
    entropy = -entropy
    return entropy
```

  - calculate the information gain：follow the IG formula

$$Gain(S, A) = E(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} E(S_v)$$

```
def information_gain(parent, left_child, right_child):
    '''
    calculates information gain of a node
```

```
    input_ : parent_list, child_list (left child and right child)
    output : float, information gain value
    '''
    left_num = len(left_child) / len(parent)
    right_num = len(right_child) / len(parent)

    child = left_num * entropy(left_child) + right_num * entropy(right_child)

    return entropy(parent) - child
```

- ▼ standardizing data

```
# Function for standardizing data
def standardScaler(feature_array):
    num = feature_array.shape[1] # total number of columns
    for i in range(num): # iterating through each column
        feature = feature_array[:, i]
        mean = feature.mean() # mean stores mean value for the column
        std = feature.std() # std stores standard deviation value for the column
        feature_array[:, i] = (feature_array[:, i] - mean) / std # standard scali
ng of each element of the column
    return feature_array
```

▼ Decision Tree

- ▼ Node in the tree：to record the information of a node in the decision tree

```
class Node:
    '''
    define the node in the decistion tree
    '''
    def __init__(self, feature=None, threshold=None, data_left=None, data_right=N
one, gain=None, value=None):
        self.feature = feature
        self.threshold = threshold
        self.data_left = data_left
        self.data_right = data_right
        self.gain = gain
        self.value = value
```

- ▼ choose the feature to split the data by the one with highest information gain

```
class DecisionTree:
    '''
    implementing decisicion tree

    '''
    def __init__(self, min_samples_split=2, max_depth=7):
      self.min_samples_split = min_samples_split
      self.max_depth = max_depth
      self.root = None

    def _best_split(self, X, y):
      '''
      calculates the best split for given features and target

      input_ : X = features, y = target
      output : best_split (dict)
      '''
      best_split = {}
      best_info_gain = -1
      n_rows, n_cols = X.shape

      # For every dataset feature
      for f_idx in range(n_cols):
          X_curr = X[:, f_idx]
          # For every unique value of that feature
          for threshold in np.unique(X_curr):
              # Construct a dataset and split it to the left and right parts
              # Left part includes records lower or equal to the threshold
              # Right part includes records higher than the threshold
              df = np.concatenate((X, y.reshape(1, -1).T), axis=1)
              left = np.array([row for row in df if row[f_idx] <= threshold])
              right = np.array([row for row in df if row[f_idx] > threshold])

              # check if data in the subset
              if len(left ) <= 0:
                continue
              if len(right ) <= 0:
                continue

              # Obtain the value of the target variable for subsets
              y = df[:, -1]
              left = left [:, -1]
              right = right [:, -1]

              # Caclulate the information gain and save the split parameters
              # if the current split if better then the previous best
              gain = information_gain(y, y_left, y_right)
              if gain > best_info_gain:
                  best_split = {
                      'feature_index': f_idx,
                      'threshold': threshold,
                      'df_left': left ,
```

```
                    'df_right': right ,
                    'gain': gain
                }
                best_info_gain = gain
        return best_split

    def _build(self, X, y, depth=0):
        '''
        build a decision tree

        input_ : X = features, y = target, depth
        output : node
        '''
        n_rows, n_cols = X.shape

        # Check to see if a node should be leaf node
        if n_rows >= self.min_samples_split and depth <= self.max_depth:
            # Get the best split
            best = self._best_split(X, y)
            # If the split isn't pure
            if best['gain'] > 0:
                # Build a tree on the left
                left = self._build(
                    X=best['df_left'][:, :-1],
                    y=best['df_left'][:, -1],
                    depth=depth + 1
                )
                right = self._build(
                    X=best['df_right'][:, :-1],
                    y=best['df_right'][:, -1],
                    depth=depth + 1
                )
                return Node(
                    feature=best['feature_index'],
                    threshold=best['threshold'],
                    data_left=left,
                    data_right=right,
                    gain=best['gain']
                )
        # Leaf node - value is the most common target value
        return Node(
            value=Counter(y).most_common(1)[0][0]
        )

    def fit(self, X, y):
        '''
        Train with given features and target

        input_ : X = features, y = target
        output : //
        '''
        # Call a recursive function to build the tree
        self.root = self._build(X, y)
```

- prediction

```
def _predict(self, x, tree):
      '''
      classify a single test data

      input_ : x (one input data)
      output : class (prediction)
      '''
      # Leaf node
      if tree.value != None:
          return tree.value
      feature_value = x[tree.feature]

      # classification
      if feature_value <= tree.threshold:
        return self._predict(x=x, tree=tree.data_left)

      else:
        return self._predict(x=x, tree=tree.data_right)

    def predict(self, testing_data):
      '''
      classify all data

      :param X: np.array, features
      :return: np.array, predicted classes
      '''
      # Call the _predict() function for every observation
      return [self._predict(entry, self.root) for entry in testing_data]
```

- Data Observation

  1. fill missing data with the mean of the feature

```
#  print(df.head(5))
df.head(n = 10).style.background_gradient(cmap = "Purples_r")
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
```

```
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         1023 non-null   float64
 1   volatile_acidity      1023 non-null   float64
 2   citric_acid           1023 non-null   float64
 3   residual_sugar        1023 non-null   float64
 4   chlorides             1023 non-null   float64
 5   free_sulfur_dioxide   1023 non-null   float64
 6   total_sulfur_dioxide  1023 non-null   float64
 7   density               1023 non-null   float64
 8   pH                    1023 non-null   float64
 9   sulphates             1023 non-null   float64
 10  alcohol               1023 non-null   float64
dtypes: float64(11)
memory usage: 88.0 KB
```

2. found that every element is unique, so i choose one to fill that missing dat

```
[96] print(train_x['Phrase'].head(10))

0                              going to a house party and
1                                         a grand picture
2                                     lightweight meaning
3                                         most unpleasant
4     You can see the would-be surprises coming a mi...
5     this too-extreme-for-TV rendition of the notor...
6                       wickedly undramatic central theme
7     ... a fascinating curiosity piece -- fascinati...
8               fallible human beings , not caricatures
9     is so prolonged and boring it is n't even clos...
Name: Phrase, dtype: object
```

```
print(train_y['Sentiment'].describe())

count    124848.000000
mean          2.063581
std           0.893844
min           0.000000
25%           2.000000
50%           2.000000
75%           3.000000
max           4.000000
Name: Sentiment, dtype: float64
```

```
[98] print(train_y['Sentiment'].value_counts())
```

```
2    63665
3    26342
1    21818
4     7365
0     5658
Name: Sentiment, dtype: int64
```

```
[99] print(train_y['Sentiment'].value_counts()/train_y['Sentiment'].count())
```

```
2    0.509940
3    0.210993
1    0.174757
4    0.058992
0    0.045319
Name: Sentiment, dtype: float64
```

```
[100] temp_df = train_x.isnull().sum().reset_index()
      temp_df['Percentage of Null Values'] = temp_df[0]/len(train_x)*100
      temp_df.columns = ['Column Name', 'Number of Null Values','Percentage of Null Values']
      temp_df
```

	Column Name	Number of Null Values	Percentage of Null Values
0	Phrase	0	0.0

```
[101] train_x.describe().T.style.background_gradient(cmap = "magma")
```

	count	unique	top	freq
Phrase	124848	124847	going to a house party and	2

Data Observation Visualization

1. dataset1

- missing data(fill with mean value)

	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfur_dioxide	density	pH	sulphates	alcohol
0	7.000000	0.230000	0.400000	1.600000	0.063000	nan	67.000000	0.995200	3.500000	0.630000	11.100000
1	7.800000	0.600000	0.260000	2.000000	0.080000	31.000000	131.000000	0.996220	nan	0.520000	9.900000
2	9.700000	0.690000	0.320000	2.500000	0.088000	22.000000	91.000000	0.997900	3.290000	0.620000	10.100000
3	12.000000	0.380000	0.560000	2.100000	0.093000	6.000000	24.000000	0.999250	3.140000	0.710000	10.900000
4	6.400000	0.640000	0.210000	1.800000	0.081000	14.000000	31.000000	0.996890	3.590000	0.660000	nan
5	7.400000	0.350000	0.330000	2.400000	0.068000	9.000000	26.000000	0.994700	nan	0.600000	11.900000
6	6.900000	0.360000	0.250000	2.400000	0.098000	5.000000	16.000000	0.996400	3.410000	0.600000	10.100000
7	7.500000	0.420000	0.310000	1.600000	0.080000	nan	42.000000	0.997800	3.310000	0.640000	9.000000
8	7.000000	0.745000	0.120000	1.800000	0.114000	nan	64.000000	0.995880	3.220000	0.590000	9.500000
9	6.900000	0.540000	0.040000	3.000000	0.077000	7.000000	27.000000	0.998700	3.690000	0.910000	9.400000

	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfur_dioxide	density	pH	sulphates	alcohol
0	7.000000	0.230000	0.400000	1.600000	0.063000	15.920807	67.000000	0.995200	3.500000	0.630000	11.100000
1	7.800000	0.600000	0.260000	2.000000	0.080000	31.000000	131.000000	0.996220	3.308632	0.520000	9.900000
2	9.700000	0.690000	0.320000	2.500000	0.088000	22.000000	91.000000	0.997900	3.290000	0.620000	10.100000
3	12.000000	0.380000	0.560000	2.100000	0.093000	6.000000	24.000000	0.999250	3.140000	0.710000	10.900000
4	6.400000	0.640000	0.210000	1.800000	0.081000	14.000000	31.000000	0.996890	3.590000	0.660000	10.445009
5	7.400000	0.350000	0.330000	2.400000	0.068000	9.000000	26.000000	0.994700	3.308632	0.600000	11.900000
6	6.900000	0.360000	0.250000	2.400000	0.098000	5.000000	16.000000	0.996400	3.410000	0.600000	10.100000
7	7.500000	0.420000	0.310000	1.600000	0.080000	15.920807	42.000000	0.997800	3.310000	0.640000	9.000000
8	7.000000	0.745000	0.120000	1.800000	0.114000	15.920807	64.000000	0.995880	3.220000	0.590000	9.500000
9	6.900000	0.540000	0.040000	3.000000	0.077000	7.000000	27.000000	0.998700	3.690000	0.910000	9.400000

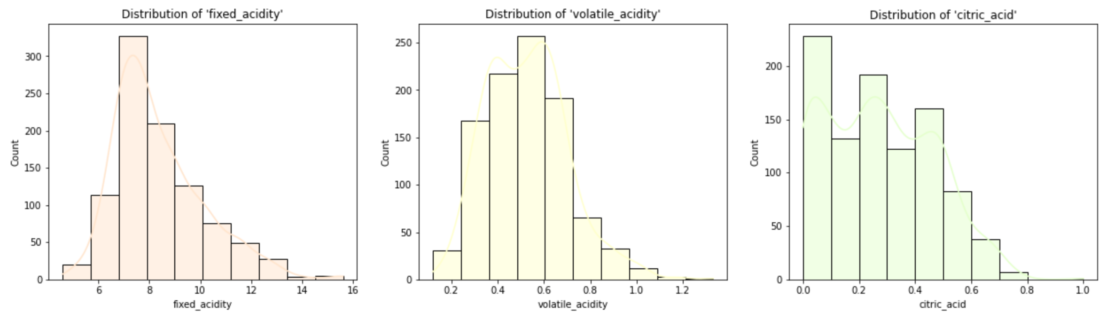
	count	mean	std	min	25%	50%	75%	max
fixed_acidity	1023.000000	8.373800	1.719035	4.600000	7.200000	8.100000	9.200000	15.600000
volatile_acidity	1023.000000	0.526118	0.172922	0.120000	0.392500	0.526118	0.635000	1.330000
citric_acid	1023.000000	0.274216	0.187548	0.000000	0.110000	0.274216	0.420000	1.000000
residual_sugar	1023.000000	2.510010	1.246486	1.200000	1.900000	2.200000	2.600000	15.400000
chlorides	1023.000000	0.087151	0.042649	0.012000	0.071000	0.080000	0.090000	0.610000
free_sulfur_dioxide	1023.000000	15.920807	9.871617	1.000000	8.000000	15.000000	21.000000	66.000000
total_sulfur_dioxide	1023.000000	45.801125	32.572508	6.000000	22.000000	38.000000	58.000000	289.000000
density	1023.000000	0.996776	0.001851	0.990200	0.995700	0.996776	0.997800	1.003690
pH	1023.000000	3.308632	0.148922	2.740000	3.210000	3.308632	3.390000	4.010000
sulphates	1023.000000	0.663580	0.172007	0.370000	0.560000	0.630000	0.730000	2.000000
alcohol	1023.000000	10.445009	1.018217	8.400000	9.600000	10.300000	11.000000	14.000000

```python
fig, axes = plt.subplots(1, 3, figsize = (20, 5))

sns.histplot(ax = axes[0], x = train_x["fixed_acidity"],
                           bins = 10,
                           kde = True,
                           cbar = True,
                           color = "#FFE5CC").set(title = "Distribution of 'fixed_acidity'");

sns.histplot(ax = axes[1], x = train_x["volatile_acidity"],
                           bins = 10,
                           cbar = True,
                           kde = True,
                           color = "#FFFFCC").set(title = "Distribution of 'volatile_acidity'");

sns.histplot(ax = axes[2], x = train_x["citric_acid"],
                           bins = 10,
                           kde = True,
                           cbar = True,
                           color = "#E5FFCC").set(title = "Distribution of 'citric_acid'");
```

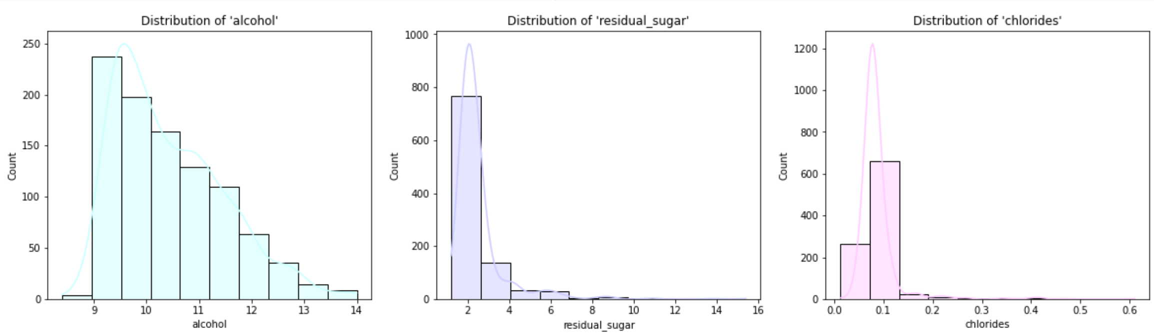



```python
fig, axes = plt.subplots(1, 3, figsize=(20, 5))

sns.histplot(ax = axes[0], x = train_x["alcohol"],
                           bins = 10,
                           kde = True,
                           cbar = True,
                           color = "#CCFFFF").set(title = "Distribution of 'alcohol'");

sns.histplot(ax = axes[1], x = train_x["residual_sugar"],
                           bins = 10,
                           kde = True,
                           cbar = True,
                           color = "#CCCCFF").set(title = "Distribution of 'residual_sugar'");

sns.histplot(ax = axes[2], x = train_x["chlorides"],
                           bins = 10,
                           kde = True,
                           cbar = True,
                           color = "#FFCCFF").set(title = "Distribution of 'chlorides'");
```



```
# Checking for outliers
plt.figure(figsize = (20, 8))
for i in range (len(train_x.columns)):
    plt.subplot(2, 6, i+1)
    sns.boxplot(x = train_x.iloc[:, i])
    plt.xlabel(train_x.columns[i], size = 12)
```