資料科學

Data Science

作業五 HW5

電機所 R11921038 江讀晉

2022/12/9

Problem 1. Decision Tree split by Gini Index

1. Every Gini index of candidates

Depth	Feature	Feature value		Gini index
0	**Car Type**	**Family**	**Sports, Luxury**	**0.2769**
	Shirt Size	Small, Medium	Extra Large, Large	0.3542
	Car Type	Sports	Family, Luxury	0.3690
	Shirt Size	Extra Large	Small, Medium, Large	0.3938
	Shirt Size	Medium	Small, Large, Extra Large	0.4198
	Gender	M	F	0.45
	Shirt Size	Small, Extra Large	Medium, Large	0.4727
	Shirt Size	Large	Small, Medium, Extra Large	0.4750
	Car Type	Luxury	Family, Sports	0.4791
	Shirt Size	Small	Medium, Large, Extra Large	0.48
	Shirt Size	Small, Large	Medium, Extra Large	0.4949
1	**Shirt Size**	**Large, Extra Large**	**Small, Medium**	**0.1231**
	Shirt Size	Extra Large	Small, Medium, Large	0.2517
	Shirt Size	Medium	Small, Large, Extra Large	0.3077
	Shirt Size	Large	Small, Medium, Extra Large	0.3487
	Shirt size	Small	Medium, Large, Extra Large	0.3692
	Car Type	Sports	Luxury	0.3919
	Gender	M	F	0.4154
	Shirt Size	Small, Extra Large	Medium, Large	0.4154
	Shirt Size	Small, Large	Medium, Extra Large	0.4249
2	**Car Type**	**Sports**	**Luxury**	**0.2**
	Gender	M	F	0.2667
	Shirt Size	Large	Extra Large	0.2667
3	**Shirt Size**	**Large**	**Extra Large**	**0**

Table 1. Gini index of each candidate. The bold texts are the selected rules at each step.

2. Decision tree

Gini index = 0.2769
C0: 11 / C1: 9

Customer ID	Gender	Car Type	Shirt Size	Class	Customer ID	Gender	Car Type	Shirt Size	Class
1	M	Family	Small	C0	11	F	Luxury	Large	C0
2	M	Family	Medium	C0	12	M	Sports	Medium	C1
3	M	Family	Extra Large	C0	13	M	Sports	Large	C1
4	M	Sports	Extra Large	C0	14	M	Sports	Medium	C1
5	M	Family	Large	C0	15	F	Sports	Small	C1
6	M	Family	Extra Large	C0	16	F	Luxury	Small	C1
7	M	Luxury	Extra Large	C0	17	F	Luxury	Small	C1
8	F	Family	Small	C0	18	F	Sports	Medium	C1
9	F	Family	Medium	C0	19	F	Luxury	Medium	C1
10	F	Luxury	Large	C0	20	F	Luxury	Medium	C1

Car Type: (Family)

Leaf Node
C0: 7

Customer ID	Gender	Car Type	Shirt Size	Class
1	M	Family	Small	C0
2	M	Family	Medium	C0
3	M	Family	Extra Large	C0
5	M	Family	Large	C0
6	M	Family	Extra Large	C0
8	F	Family	Small	C0
9	F	Family	Medium	C0

Car Type: (Sports, Luxury)

Gini index = 0.1231
C0: 4 / C1: 9

Customer ID	Gender	Car Type	Shirt Size	Class
4	M	Sports	Extra Large	C0
7	M	Luxury	Extra Large	C0
10	F	Luxury	Large	C0
11	F	Luxury	Large	C0
12	M	Sports	Medium	C1
13	M	Sports	Large	C1
14	M	Sports	Medium	C1
15	F	Sports	Small	C1
16	F	Luxury	Small	C1
17	F	Luxury	Small	C1
18	F	Sports	Medium	C1
19	F	Luxury	Medium	C1
20	F	Luxury	Medium	C1

Shirt Size: (Medium, Small)

Leaf Node
C1: 7

Customer ID	Gender	Car Type	Shirt Size	Class
12	M	Sports	Medium	C1
14	M	Sports	Medium	C1
15	F	Sports	Small	C1
16	F	Luxury	Small	C1
17	F	Luxury	Small	C1
18	F	Sports	Medium	C1
19	F	Luxury	Medium	C1
20	F	Luxury	Medium	C1

Shirt Size: (Extra Large, Large)

Gini index = 0.2
C0: 4 / C1: 1

Customer ID	Gender	Car Type	Shirt Size	Class
4	M	Sports	Extra Large	C0
7	M	Luxury	Extra Large	C0
10	F	Luxury	Large	C0
11	F	Luxury	Large	C0
13	M	Sports	Large	C1

Car Type: (Sports)

Gini index = 0
C0: 1 / C1: 1

Customer ID	Gender	Car Type	Shirt Size	Class
4	M	Sports	Extra Large	C0
13	M	Sports	Large	C1

Car Type: (Luxury)

Leaf Node
C0: 3

Customer ID	Gender	Car Type	Shirt Size	Class
7	M	Luxury	Extra Large	C0
10	F	Luxury	Large	C0
11	F	Luxury	Large	C0

Shirt Size: (Extra Large)

Leaf Node
C0: 1

Customer ID	Gender	Car Type	Shirt Size	Class
4	M	Sports	Extra Large	C0

Shirt Size: (Large)

Leaf Node
C1: 1

Customer ID	Gender	Car Type	Shirt Size	Class
13	M	Sports	Large	C1

Fig. 1 Decision tree with node information

3. The programming implementation

```python
class DecisionTree:
    def __init__(self, data, columns = None, depth = 0, max_depth = 10):
        self.data = data
        self.columns = columns
        self.depth = depth
        self.max_depth = max_depth
        self.left = None
        self.right = None
        self.gini_candidates = {'Feature': [], 'Value': [], 'Gini Index': []}
        self.best_gini = None
        self.split_feature = None
        self.split_value = None
        self.target = None
        self.build_tree()

    def build_tree(self):
        # If the data is empty, return
        if self.data.empty:
            return

        # If the data is pure, return
        if len(self.data[self.columns[-1]].unique()) == 1:
            self.target = self.data[self.columns[-1]].unique()[0]
            return

        # If the depth is greater than the max depth, return
        if self.depth >= self.max_depth:
            self.target = self.data[self.columns[-1]].value_counts().idxmax()
            return

        # Get the best split feature and value
        self.best_gini, self.split_feature, self.split_value = self.get_best_split()

        # Split the data
        left_data = self.data[self.data[self.split_feature].isin(self.split_value)]
        right_data = self.data[~self.data[self.split_feature].isin(self.split_value)]

        # Display the decision tree
        self.display_tree(left_data, right_data)

        # Save the children nodes to csv file
        left_data.to_csv(f'left_node_{self.depth}.csv', index = False)
        right_data.to_csv(f'right_node_{self.depth}.csv', index = False)

        # Build the left and right subtrees
        self.left = DecisionTree(left_data, self.columns, self.depth + 1, self.max_depth)
        self.right = DecisionTree(right_data, self.columns, self.depth + 1, self.max_depth)


    def display_tree(self, left_data, right_data):
        print('Depth:', self.depth)

        candidates = pd.DataFrame(self.gini_candidates)
        candidates = candidates.sort_values(by = 'Gini Index', ascending = True)
        print(f'Gini candidates:\n{candidates}')

        split = pd.DataFrame()
        split['Split Feature'] = [self.split_feature]
        split['Split Value'] = [self.split_value]
        split['Gini Index'] = [self.best_gini]

        print(f'Split parameter:\n{split}')
        print(f'Left node:\n{left_data}')
        print(f'Right node:\n{right_data}\n')

    def get_best_split(self):
        # Get the best split feature and value
```

```python
        best_split_feature = None
        best_split_value = None
        best_gini = 1.0

        # Loop through all the features
        for feature in self.columns[1:-1]:
            # Get the unique values of the feature
            values = self.data[feature].unique()
            # print('ori', values)

            # Build up the combination of the values
            new_values = []
            for i in range(len(values) // 2):
                new_values.extend(list(combinations(values, i + 1)))

            # Loop through all the values
            for value in new_values:
                # Split the data
                left_data = self.data[self.data[feature].isin(value)]
                right_data = self.data[~(self.data[feature].isin(value))]

                # Calculate the gini index
                gini = self.get_gini(left_data, right_data)
                self.gini_candidates['Feature'].append(feature)
                self.gini_candidates['Value'].append(value)
                self.gini_candidates['Gini Index'].append(round(gini, 4))

                # If the gini index is less than the best gini index, update the best gini index
                if gini < best_gini:
                    best_gini = round(gini, 4)
                    best_split_feature = feature
                    best_split_value = value

        # Return the best split feature and value
        return best_gini, best_split_feature, best_split_value

    def get_gini(self, left_data, right_data):
        # Calculate the gini index
        gini_left = 0.0
        gini_right = 0.0

        # Get the target categories
        target = self.data[self.columns[-1]].unique()

        # Calculate the gini index
        if len(left_data) > 0:
            for t in target:
                gini_left += (len(left_data[left_data[self.columns[-1]] == t]) / len(left_data)) ** 2
        gini_left = 1 - gini_left

        if len(right_data) > 0:
            for t in target:
                gini_right += (len(right_data[right_data[self.columns[-1]] == t]) / len(right_data)) ** 2
        gini_right = 1 - gini_right

        gini = (len(left_data) / len(self.data)) * gini_left + (len(right_data) / len(self.data)) * gini_right

        return gini


if __name__ == '__main__':
    # Read in the data
    df = pd.read_csv('data.csv')
    print(df)

```

```
    # Get the column names
    columns = df.columns
    print(columns)

    # Get the unique feature categories and the target categories
    gender = df[columns[1]].unique()
    car = df[columns[2]].unique()
    shirt = df[columns[3]].unique()
    target = df[columns[4]].unique()

    # Build the decision tree
    tree = DecisionTree(df, columns)
```

Problem 2. Naïve Bayes Classifier

Given tuple: (Gender=M, Car Type=Sports, Shirt Size=Medium)

$$A: features\ of\ Gender, Car\ Type\ and\ Shirt\ Size$$

$$P(A|C0) = P(Gender|C0) \cdot P(Car\ Type|C0) \cdot P(Shirt\ Size|C0) = \frac{7}{11} \cdot \frac{1}{11} \cdot \frac{2}{11} = \frac{14}{1331} = 0.0105$$

$$P(A|C1) = P(Gender|C1) \cdot P(Car\ Type|C1) \cdot P(Shirt\ Size|C1) = \frac{3}{9} \cdot \frac{5}{9} \cdot \frac{5}{9} = \frac{25}{243} = 0.1029$$

$$P(A|C0)P(C0) = \frac{14}{1331} \cdot \frac{11}{20} = 0.0058$$

$$P(A|C1)P(C1) = \frac{25}{243} \cdot \frac{9}{20} = 0.0463$$

As $P(A|C0)P(C0) < P(A|C1)P(C1)$, the tuple (Gender=M, Car Type=Sports, Shirt Size=Medium) should be classified into class C1.

Problem 3. SVM (Support Vector Machine)

Positive samples, y = 1: (4, 3), (4, 8), (7, 2) Negative samples, y = -1: (-1, -2), (-1, 3), (2, -1), (2, 1)

1. Objectives and constraints

The objective with the constraint is to

$$\text{maximize } \frac{\mathbf{2}}{\|\boldsymbol{w}\|^2}$$

$$\text{subject to } \boldsymbol{y_i}(\boldsymbol{w^T x_i} + \boldsymbol{b}) - \mathbf{1} \ \geq \mathbf{0}, \forall \boldsymbol{x_i}$$

Hence, the hinge loss with the regularization of the weights is used.

$$J = \lambda \|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max\left(0, 1 - y_i(w \cdot x_i + b)\right)$$

2. Support vectors

The two support vectors are **(4, 3)** and **(2, 1)**. These two support vectors are highlighted in the Fig. 3.

3. Computing progress

Based on the loss function, the gradient of weight and bias can be computed.

$$if\ y_i(w \cdot x_i + b) \geq 1, then\ J_i = \lambda\|w\|^2 \quad \therefore \frac{\partial \mathrm{J_i}}{\partial w_k} = 2\lambda w_k$$

$$else\ J_i = \lambda\|w\|^2 + 1 - y_i(w \cdot x_i + b) \quad \therefore \begin{cases} \dfrac{\partial \mathrm{J_i}}{\partial w_k} = 2\lambda w_k - y_i \cdot x_i \\ \dfrac{\partial \mathrm{J_i}}{\partial b} = -y_i \end{cases}$$

The following is my programming implementation.

```python
class LinearSVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.1, n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None
        self.w_list = []
        self.b_list = []

    def fit(self, x, y):
        self.w = np.zeros(x.shape[1])
        self.b = 0

        self.w_list = [list(self.w)]
        self.b_list = [self.b]
        for _ in range(self.n_iters):
            for i in range(x.shape[0]):
                if y[i] * (np.dot(x[i], self.w) + self.b) >= 1:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x[i], y[i]))
                    self.b -= self.lr * (-y[i])

            if np.linalg.norm(self.w - self.w_list[-1]) < 1e-4:
                break
            else:
                self.w_list.append(list(self.w))
                self.b_list.append(self.b)

if __name__ == '__main__':
    # Construct the data
    data = np.array([[4, 3], [4, 8], [7, 2], [-1, -2], [-1, 3], [2, -1], [2, 1]])
    target = np.array([1, 1, 1, -1, -1, -1, -1])

    # Train the model
    svm = LinearSVM(n_iters=10000)
    svm.fit(data, target)
```

4. Weight, bias and hyperplane

Based the codes above, the weight and the bias are converged to

$$w = \begin{bmatrix} 0.4999023 \\ 0.5003976 \end{bmatrix}, b = -2.5039999$$

If the weight and the bias are round to 1 decimal place, then

$$\boldsymbol{w} = \begin{bmatrix} \mathbf{0.5} \\ \mathbf{0.5} \end{bmatrix}, \boldsymbol{b} = -\mathbf{2.5}$$

The curves of the weight and the bias in every iteration are shown in the Fig. 2.

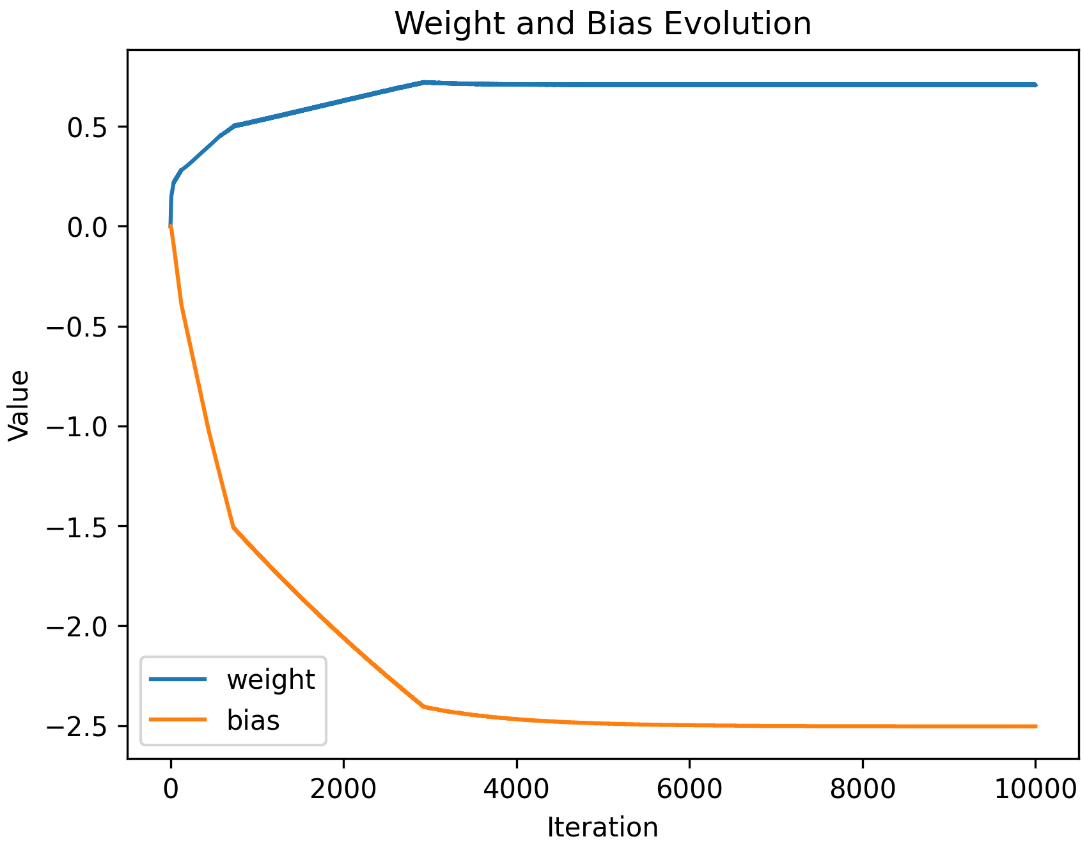

Fig. 2 The evolution of the weight and the bias

The decision boundary or hyperplane is

$$\begin{aligned} \boldsymbol{y} &= w^T x + b \\ &= w_1 x_1 + w_2 x_2 + b \\ &= \mathbf{0.5}\boldsymbol{x_1} + \mathbf{0.5}\boldsymbol{x_2} - \mathbf{2.5} \end{aligned}$$

To illustrate the results, the data points with labels, hyperplane (decision boundary), two boundaries of margin and support vectors are manifested in the Fig. 3.

Note that HW5-P3 is a hard SVM and there are exactly two support vectors on the boundaries of margin.

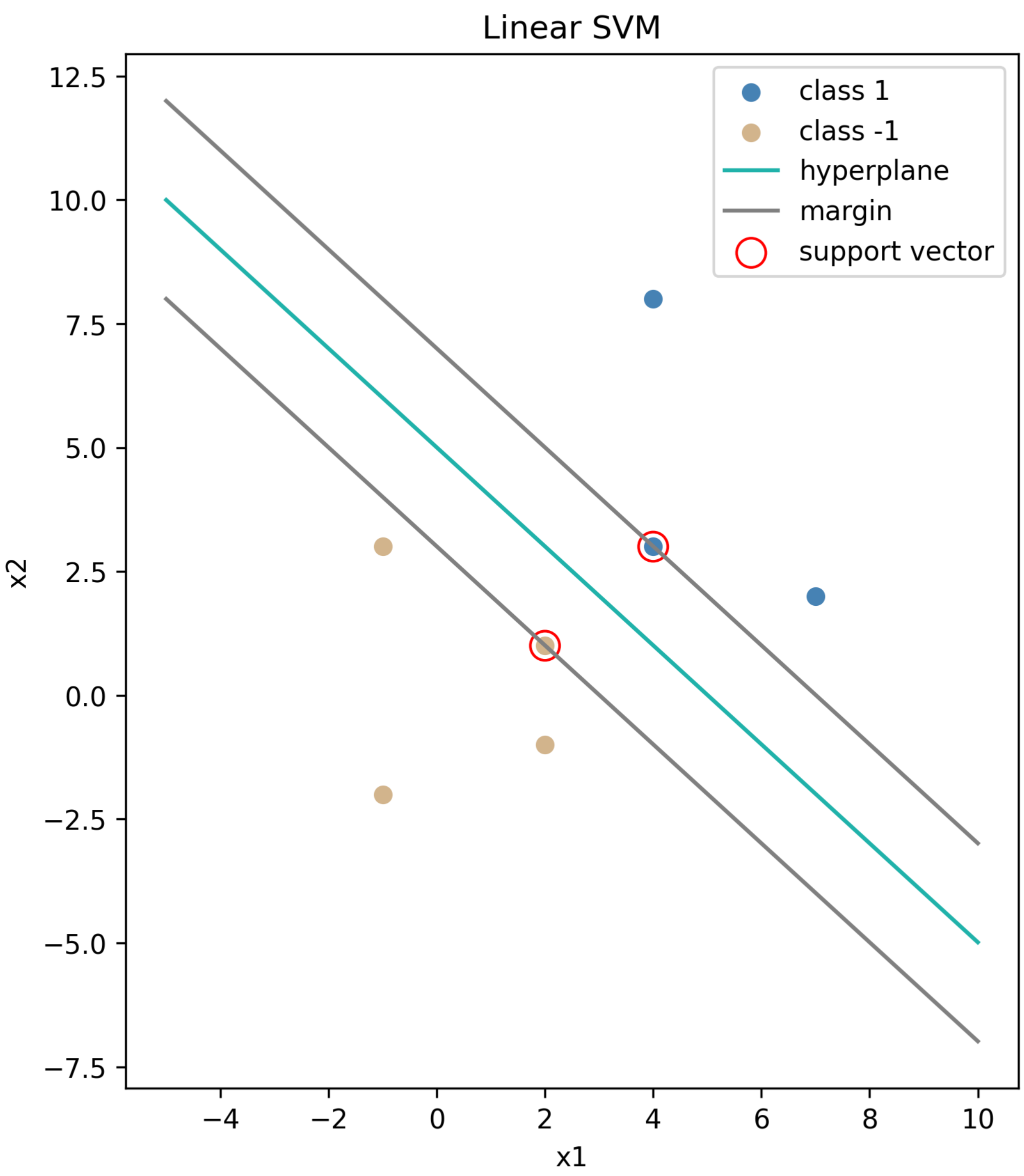

Fig. 3 The result of the SVM classifier