W7_03_04_2024

Lê Thị Minh Anh

April 3, 2024

1 Xây dựng cây quyết định với ID3

1.1 Cơ sở lý thuyết

Mô hình cây quyết định là một mô hình được sử dụng khá phổ biến và hiệu quả trong cả hai lớp bài toán phân loại và dự báo của học có giám sát. Một cây quyết định là một cấu trúc thuật toán giống như flowchart, với mỗi một nút (node) là một phép so sánh, nhằm thực hiện thử một đặc trưng nào đó. Mỗi một nhánh (branch, edge, etc.) thể hiện kết quả của phép so sánh đó, và mỗi một ngọn, tức nút cuối (leaf node), biểu diễn kết quả phân loại của thuật toán, sau khi tính . Đường đi từ gốc đến ngọn thể hiện luật phân loại (classification rules).

Thuật toán ID3 (viết tắt của Iterative Dichotomiser 3) là một giải thuật khá lâu đời được tạo ra bởi Ross Quinlan nhằm xây dựng cây quyết định phù hợp từ một bộ dữ liệu.

Xét một bài toán với C class khác nhau. Giả sử ta đang làm việc với một non-leaf node với các điểm dữ liệu tạo thành một tập S với số phần tử là |S| = N . Giả sử thêm rằng trong số N điểm dữ liệu này, N_c , c = 1 , 2 , ... , C điểm thuộc vào class c. Xác suất để mỗi điểm dữ liệu rơi vào một class c được xấp xỉ bằng $\frac{N_c}{N}$ (maximum likelihood estimation). Như vậy, entropy tại node này được tính bởi:

$$Entropy(S) = -\sum_{c=1}^{C} \frac{N_c}{N} \log_2 \frac{N_c}{N} \quad (1)$$

Tiếp theo, giả sử thuộc tính được chọn là x. Dựa trên x , các điểm dữ liệu trong S được phân ra thành K child node $S_1, S_2, ..., S_K$ với số điểm trong mỗi child node lần lượt là $m_1, m_2, ..., m_K$. Ta định nghĩa

$$Entropy(S, x) = \sum_{k=1}^{K} \frac{m_k}{N} Entropy(S_k) \quad (2)$$

là tổng có trọng số entropy của mỗi child node-được tính tương tự như (1). Việc lấy trọng số này là quan trọng vì các node thường có số lượng điểm khác nhau.

Tiếp theo, ta định nghĩa information gain dựa trên thuộc tính x:

$$Gain(S, x) = Entropy(S) - Entropy(S, x)$$

Trong ID3, tại mỗi node, thuộc tính được chọn được xác định dựa trên:

$$x^* = \arg\max_x Gain(S, x) = \arg\min_x Entropy(S, x) \quad (3)$$

1.2 Ví dụ

Bảng dữ liệu này mô tả mối quan hệ giữa thời tiết trong 14 ngày (bốn cột đầu, không tính cột id) và việc một đội bóng có chơi bóng hay không (cột cuối cùng).

id	outlook	temperature	humidity	windy	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no

Có bốn thuộc tính thời tiết:

1. Outlook nhận một trong ba giá trị: sunny, overcast, rain.
2. Temperature nhận một trong ba giá trị: hot, cool, mild.
3. Humidity nhận một trong hai giá trị: high, normal.
4. Wind nhận một trong hai giá trị: weak, strong.

Chúng ta sẽ cùng tìm thứ tự các thuộc tính bằng thuật toán ID3.

Tập dữ liệu hiện tại có 9 kết quả Yes và 5 kết quả No, ta kí hiệu là S: [9+,5-].

Chọn nút gốc của cây quyết định:

Entropy của tập dữ liệu hiện tại:

$$Entropy(S) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.94$$

Xét thuộc tính Outlook, thuộc tính này nhận 3 giá trị là Sunny, Overcast, Rain. Ứng với mỗi thuộc tính, ta có:

- S_{Sunny}: [2+,3-] (có nghĩa là trong tập dữ liệu hiện tại (S), có 2 kết quả Yes và 3 kết quả No tại Outlook = Sunny). Tương tự:
- $S_{Overcast}$: [4+,0-]

- S_{Rain}: [3+,2-]

Entropy tại node này nếu chọn thuộc tính Outlook là:

$$Entropy(S, Outlook) = \frac{5}{14} \times Entropy(S_{Sunny}) + \frac{4}{14} \times Entropy(S_{Overcast}) + \frac{5}{14} \times Entropy(S_{Rain}) = 0.694$$

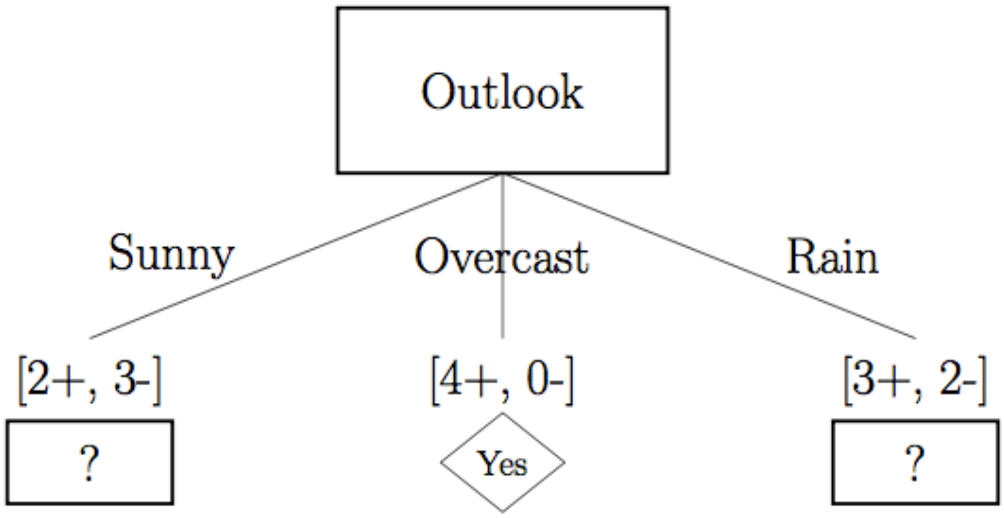
Information gain tương ứng với thuộc tính Outlook:

$$Gain(S, Outlook) = Entropy(S) - Entropy(S, Outlook) = 0.246$$

Tương tự, ta tính được information gain tương ứng với các thuộc tính còn lại:

- Gain(S, Temperature) = 0.029
- Gain(S, Humidity) = 0.152
- Gain(S, Wind) = 0.048

Thuộc tính Outlook có Information Gain cao nhất, chọn nó làm nút gốc.



Xây dựng tiếp cây quyết định:

Sau khi chọn được nút gốc là Outlook, tiếp theo ta tính tiếp các nút tại mỗi thuộc tính của nút vừa chọn.

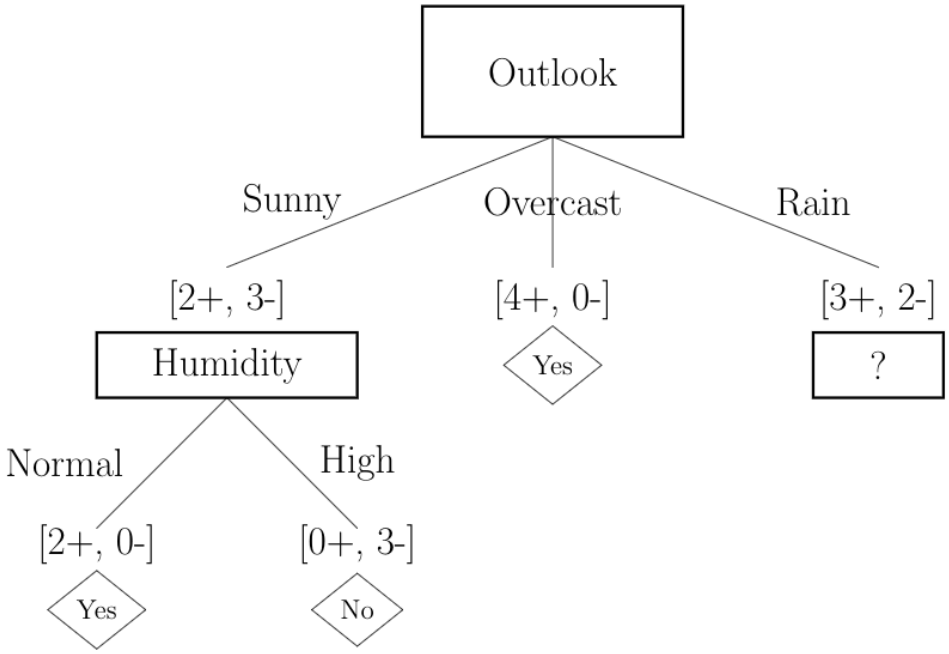
- Nhánh bên trái ứng với Outlook = Sunny, có S_{Sunny} là [2+,3-], chưa phân lớp hoàn toàn nên vẫn phải tính toán chọn nút tại đây. Tương tự cho nhánh bên phải.
- Nhánh ở giữa ứng với Outlook = Overcast, tập dữ liệu tại nhánh này đã hoàn toàn phân lớp dương với 4+ và 0-. Tại đây đã có thể quyết định, khi Outlook = Overcast thì có thể đi chơi tennis.

Bây giờ ta sẽ thực hiện tính toán với nhánh bên trái, trên tập S_{Sunny} = [2+,3-].

Hoàn toàn tương tự như cách tìm nút gốc, ta tính Information Gain cho 3 thuộc tính còn lại là Temp, Humidity và Wind (trên tập S_{Sunny}).

- Gain(S_{Sunny} , Temperature) = 0.571
- Gain(S_{Sunny} , Humidity) = 0.971
- Gain(S_{Sunny} , Wind) = 0.019

Ta thấy rằng thuộc tính Humidity cho Information Gain cao nhất, chọn nó làm nút tiếp theo cho nhánh bên trái.

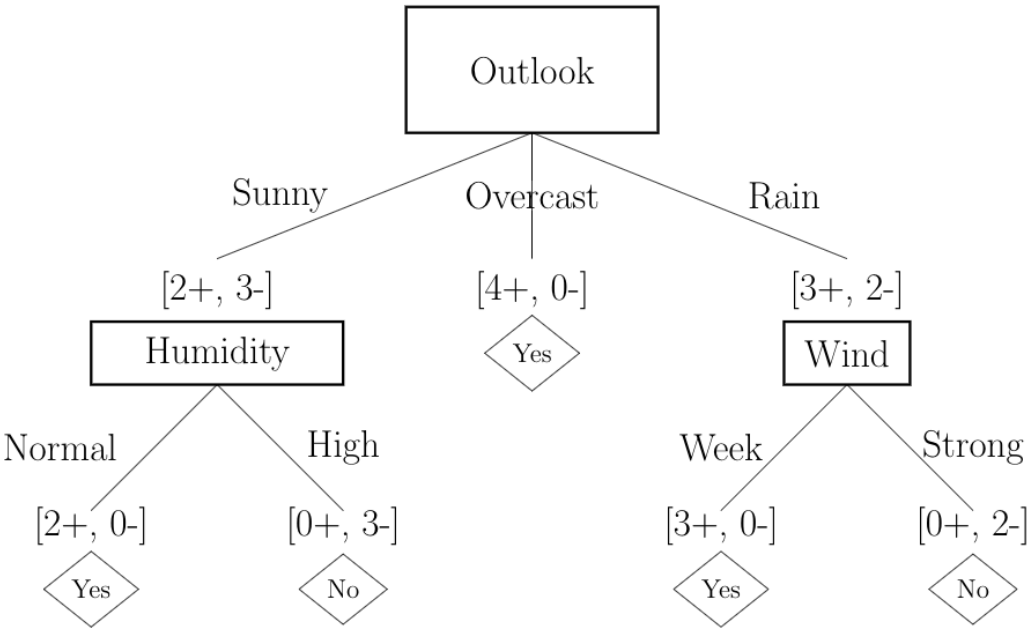


Khi đó, nhánh bên trái của nút Outlook = Sunny và Humidity = High, ta có tập dữ liệu là [0+,3-], đã phân lớp hoàn toàn, không cần phân chia nữa. Tại đây, kết luận là không đi chơi tennis. Nhánh bên phải của nút Outlook = Sunny và Humidity = Normal, ta có tập dữ liệu là [2+,0-], đã phân lớp hoàn toàn, không cần phân chia nữa. Tại đây, kết luận là đi chơi tennis.

Bây giờ ta thực hiện tính toán với nhánh bên phải, trên tập S_{Rain} = [3+,2-].

- Gain(S_{Rain} , Temperature) = 0.012
- Gain(S_{Rain} , Humidity) = 0.012
- Gain(S_{Rain} , Wind) = 0.971

Ta thấy rằng thuộc tính Wind cho Information Gain cao nhất, chọn nó làm nút tiếp theo cho nhánh bên phải.



Khi đó, nhánh bên trái của nút `Outlook` = Rain và `Wind` = Weak, ta có tập dữ liệu là [3+,0-], đã phân lớp hoàn toàn, không cần phân chia nữa. Tại đây, kết luận là đi chơi tennis. Nhánh bên phải của nút `Outlook` = Rain và `Wind` = Strong, ta có tập dữ liệu là [0+,2-], đã phân lớp hoàn toàn, không cần phân chia nữa. Tại đây, kết luận là không đi chơi tennis.

Như vậy, cây quyết định đã được xây dựng hoàn tất.

1.3 Implementing in Python

Xây dựng class TreeNode

[4]:
```
from __future__ import print_function
import numpy as np
import pandas as pd

class TreeNode(object):
    def __init__(self, ids = None, children = [], entropy = 0, depth = 0):
        self.ids = ids           # index of data in this node
        self.entropy = entropy   # entropy, will fill later
        self.depth = depth       # distance to root node
        self.split_attribute = None # which attribute is chosen, it non-leaf
        self.children = children # list of its child nodes
        self.order = None        # order of values of split_attribute in children
        self.label = None        # label of node if it is a leaf

    def set_properties(self, split_attribute, order):
        self.split_attribute = split_attribute # split at which attribute
```

```
        self.order = order # order of this node's children

    def set_label(self, label):
        self.label = label # set label if the node is a leaf
```

Hàm tính entropy dựa trên tần suất

[5]:
```
def entropy(freq):
    # remove prob 0
    freq_0 = freq[np.array(freq).nonzero()[0]]
    prob_0 = freq_0/float(freq_0.sum())
    return -np.sum(prob_0*np.log2(prob_0))
```

Xây dựng class DecisionTreeID3

[6]:
```
class DecisionTreeID3(object):
    def __init__(self, max_depth= 10, min_samples_split = 2, min_gain = 1e-4):
        self.root = None
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.Ntrain = 0
        self.min_gain = min_gain

    def fit(self, data, target):
        self.Ntrain = data.count()[0]
        self.data = data
        self.attributes = list(data)
        self.target = target
        self.labels = target.unique()

        ids = range(self.Ntrain)
        self.root = TreeNode(ids = ids, entropy = self._entropy(ids), depth = 0)
        queue = [self.root]
        while queue:
            node = queue.pop()
            if node.depth < self.max_depth or node.entropy < self.min_gain:
                node.children = self._split(node)
                if not node.children: #leaf node
                    self._set_label(node)
                queue += node.children
            else:
                self._set_label(node)

    def _entropy(self, ids):
        # calculate entropy of a node with index ids
        if len(ids) == 0: return 0
        ids = [i+1 for i in ids] # panda series index starts from 1
        freq = np.array(self.target[ids].value_counts())
```

```
        return entropy(freq)

    def _set_label(self, node):
        # find label for a node if it is a leaf
        # simply chose by major voting
        target_ids = [i + 1 for i in node.ids]  # target is a series variable
        node.set_label(self.target[target_ids].mode()[0]) # most frequent label

    def _split(self, node):
        ids = node.ids
        best_gain = 0
        best_splits = []
        best_attribute = None
        order = None
        sub_data = self.data.iloc[ids, :]
        for i, att in enumerate(self.attributes):
            values = self.data.iloc[ids, i].unique().tolist()
            if len(values) == 1: continue # entropy = 0
            splits = []
            for val in values:
                sub_ids = sub_data.index[sub_data[att] == val].tolist()
                splits.append([sub_id-1 for sub_id in sub_ids])
            # don't split if a node has too small number of points
            if min(map(len, splits)) < self.min_samples_split: continue
            # information gain
            HxS= 0
            for split in splits:
                HxS += len(split)*self._entropy(split)/len(ids)
            gain = node.entropy - HxS
            if gain < self.min_gain: continue # stop if small gain
            if gain > best_gain:
                best_gain = gain
                best_splits = splits
                best_attribute = att
                order = values
        node.set_properties(best_attribute, order)
        child_nodes = [TreeNode(ids = split,
                    entropy = self._entropy(split), depth = node.depth + 1)␣
↪for split in best_splits]
        return child_nodes

    def predict(self, new_data):
        """
        :param new_data: a new dataframe, each row is a datapoint
        :return: predicted labels for each row
        """
        npoints = new_data.count()[0]
```

```
        labels = [None]*npoints
        for n in range(npoints):
            x = new_data.iloc[n, :] # one point
            # start from root and recursively travel if not meet a leaf
            node = self.root
            while node.children:
                node = node.children[node.order.index(x[node.split_attribute])]
            labels[n] = node.label

        return labels
```

[7]:
```
df = pd.read_csv('data/weather.csv', index_col = 0)
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
tree = DecisionTreeID3(max_depth = 3, min_samples_split = 2)
tree.fit(X, y)
print(tree.predict(X))
```

```
['no', 'no', 'yes', 'yes', 'yes', 'no', 'yes', 'no', 'yes', 'yes', 'yes', 'yes',
'yes', 'no']
```

2 Xây dựng mô hình phân lớp Bayes:

2.1 Naive Bayes Classifier

Xét bài toán classification với C classes 1, 2, ...,C. Giả sử có một điểm dữ liệu x $\in R^d$. Hãy tính xác suất để điểm dữ liệu này rơi vào class c. Nói cách khác, hãy tính:

$$p(Y = c|X = x) \quad (1)$$

Biểu thức này, nếu tính được, sẽ giúp chúng ta xác định được xác suất để điểm dữ liệu rơi vào mỗi class. Từ đó có thể giúp xác định class của điểm dữ liệu đó bằng cách chọn ra class có xác suất cao nhất:

$$c = \arg\max_c p(c|x) \quad (2)$$

Biểu thức (2) thường khó được tính trực tiếp. Thay vào đó, quy tắc Bayes thường được sử dụng:

$$c = \arg\max_c p(c|x) = \arg\max_c \frac{p(x|c)p(c)}{p(x)} = \arg\max_c p(x|c)p(c) \quad (3)$$

giả sử một cách đơn giản nhất rằng các thành phần của biến ngẫu nhiên x là độc lập với nhau, nếu biết c (given c). Tức là:

$$p(x|c) = p(x_1, x_2, ..., x_d|c) = \prod_{i=1}^{d} p(x_i|c) \quad (4)$$

2.2 Các phân phối thường dùng cho $p(x_i|c)$

2.2.1 Gaussian Naive Bayes

Mô hình này được sử dụng chủ yếu trong loại dữ liệu mà các thành phần là các biến liên tục.

Với mỗi chiều dữ liệu i và một class c, x_i tuân theo một phân phối chuẩn có kỳ vọng μ_{ci} và phương sai σ_{ci}^2:

$$p(x_i|c) = p(x_i|\mu_{ci}, \sigma_{ci}^2) = \frac{1}{\sqrt{2\pi\sigma_{ci}^2}} \exp\left(-\frac{(x_i - \mu_{ci})^2}{2\sigma_{ci}^2}\right) \quad (5)$$

trong đó μ_{ci} và σ_{ci}^2 được xác định bằng Maximum Likelihood:

$$(\mu_{ci}, \sigma_{ci}^2) = \arg\max_{\mu_{ci}, \sigma_{ci}^2} \prod_{i=1}^{N} p(x_i^{(n)}|\mu_{ci}, \sigma_{ci}^2) \quad (6)$$

2.2.2 Multinomial Naive Bayes

Mô hình này chủ yếu được sử dụng trong phân loại văn bản mà feature vectors được tính bằng Bags of Words. Lúc này, mỗi văn bản được biểu diễn bởi một vector có độ dài d chính là số từ trong từ điển. Giá trị của thành phần thứ i trong mỗi vector chính là số lần từ thứ i xuất hiện trong văn bản đó.

Khi đó, $p(x_i|c)$ tỉ lệ với tần suất từ thứ i (hay feature thứ i cho trường hợp tổng quát) xuất hiện trong các văn bản của class c. Giá trị này có thể được tính bằng cách:

$$p(x_i|c) = \frac{N_{ci} + \alpha}{N_c + \alpha d} \quad (6)$$

trong đó N_{ci} là số lần từ thứ i xuất hiện trong các văn bản thuộc class c, N_c là tổng số từ trong các văn bản thuộc class c, d là số từ trong từ điển và α là một hằng số smoothing.

2.2.3 Bernoulli Naive Bayes

Trong trường hợp x_i là một biến nhị phân, ta thường giả định rằng $p(x_i|c)$ tuân theo phân phối Bernoulli:

$$p(x_i|c) = p(i|c)^{x_i}(1 - p(i|c))^{1-x_i} \quad (7)$$

trong đó $p(i|c)$ là xác suất từ thứ i xuất hiện trong các văn bản thuộc class c.

2.3 Ví dụ

2.3.1 Bắc hay Nam

Giả sử trong tập training có các văn bản d1, d2, d3, d4 như trong bảng dưới đây. Mỗi văn bản này thuộc vào 1 trong 2 classes: B (Bắc) hoặc N (Nam). Hãy xác định class của văn bản d5.

_	Document	Content	Class
Training	d1	hanoi pho chaolong hanoi	B
_	d2	hanoi buncha pho omai	B
_	d3	pho banhgio omai	B
_	d4	saigon hutiu banhbo pho	N
Test	d5	hanoi hanoi buncha hutiu	?

Bài toán này có thể được giải quyết bởi hai mô hình: Multinomial Naive Bayes và Bernoulli Naive Bayes.

Ta cần tìm p(B) và p(N):

$$p(B) = \frac{3}{4}, \quad p(N) = \frac{1}{4}$$

Tập hợp toàn bộ các từ trong văn bản, hay còn gọi là từ điển, là: V = { hanoi, pho, chaolong, buncha, omai, banhgio, saigon, hutiu, banhbo } . Tổng cộng số phần tử trong từ điển là | V | = 9.

Sử dụng Multinomial Naive Bayes, trong đó có sử dụng Laplace smoothing với $\alpha = 1$

Chúng ta lập các vector đặc trưng (feature vector) ứng với mỗi văn bản theo cách: Vector sẽ có số chiều bằng số phần tử từ điển (là 9); Ứng với mỗi văn bản dk trong tập training, xét từ có chỉ số j trong từ điển xuất hiện xj lần trong văn bản (j=1, 2, …,9), ta lập được vector đặc trưng tương ứng là: xk = (x1, x2, … , x9). Cụ thể với tập training d1, d2, d3, d4 ta có:

- Với d1: x1 = (2, 1, 1, 0, 0, 0, 0, 0, 0) (B)
- Với d2: x2 = (1, 1, 0, 1, 1, 0, 0, 0, 0) (B)
- Với d3: x3 = (0, 1, 0, 0, 1, 1, 0, 0, 0) (B)
- Với d4: x4 = (0, 1, 0, 0, 0, 0, 1, 1, 1) (N)

Với tập test:

- d5: x5 = (2, 0, 0, 1, 0, 0, 0, 1, 0)

Tổng số lượt từ (kể cả lặp lại) xuất hiện trong các văn bản thuộc lớp B là: $N_B = 11$.

Tổng số lượt từ (kể cả lặp lại) xuất hiện trong các văn bản thuộc lớp N là: $N_N = 4$.

Tính các λ_j^B, j=1, 2, …,9 - tương ứng với các từ trong từ điển.

$$\lambda_1^B = \frac{2+1+1}{11+9} = \frac{4}{20}$$

$$\lambda_2^B = \frac{1+1+1+1}{11+9} = \frac{4}{20}$$

$$\lambda_3^B = \lambda_4^B = \lambda_6^B = \frac{1+1}{11+9} = \frac{2}{20}$$

$$\lambda_5^B = \frac{1+1+1}{11+9} = \frac{3}{20}$$

$$\lambda_7^B = \lambda_8^B = \lambda_9^B = \frac{0+1}{11+9} = \frac{1}{20}$$

Tương tự, tính được các giá trị λ_j^N, j=1,2,...,9

$$\lambda_1^N = \lambda_3^N = \lambda_4^N = \lambda_5^N = \lambda_6^N = \frac{1}{13}$$

$$\lambda_2^N = \lambda_7^N = \lambda_8^N = \lambda_9^N = \frac{1+1}{4+9} = \frac{2}{13}$$

Trên tập test

$$p(B|d5) \propto p(B)\prod_{i=1}^{d} p(x_i|B) = \frac{3}{4}(\frac{4}{20})^2\frac{2}{20}\frac{1}{20} = 1.5 \times 10^{-4}$$

$$p(N|d5) \propto p(N)\prod_{i=1}^{d} p(x_i|N) = \frac{1}{4}(\frac{1}{13})^2\frac{1}{13}\frac{2}{13} = 1.75 \times 10^{-5}$$

$\Rightarrow p(x5|B) > p(x5|N) \Rightarrow d5 \in class(B)$

Implementing in Python

[8]:
```
from __future__ import print_function
from sklearn.naive_bayes import MultinomialNB
import numpy as np

# train data
d1 = [2, 1, 1, 0, 0, 0, 0, 0, 0]
d2 = [1, 1, 0, 1, 1, 0, 0, 0, 0]
d3 = [0, 1, 0, 0, 1, 1, 0, 0, 0]
d4 = [0, 1, 0, 0, 0, 0, 1, 1, 1]

train_data = np.array([d1, d2, d3, d4])
label = np.array(['B', 'B', 'B', 'N'])

# test data
d5 = np.array([[2, 0, 0, 1, 0, 0, 0, 1, 0]])

## call MultinomialNB
clf = MultinomialNB()
# training
clf.fit(train_data, label)
```

```
# test
print('Predicting class of d5:', str(clf.predict(d5)[0]))
print('Probability of d5 in each class:', clf.predict_proba(d5))
```

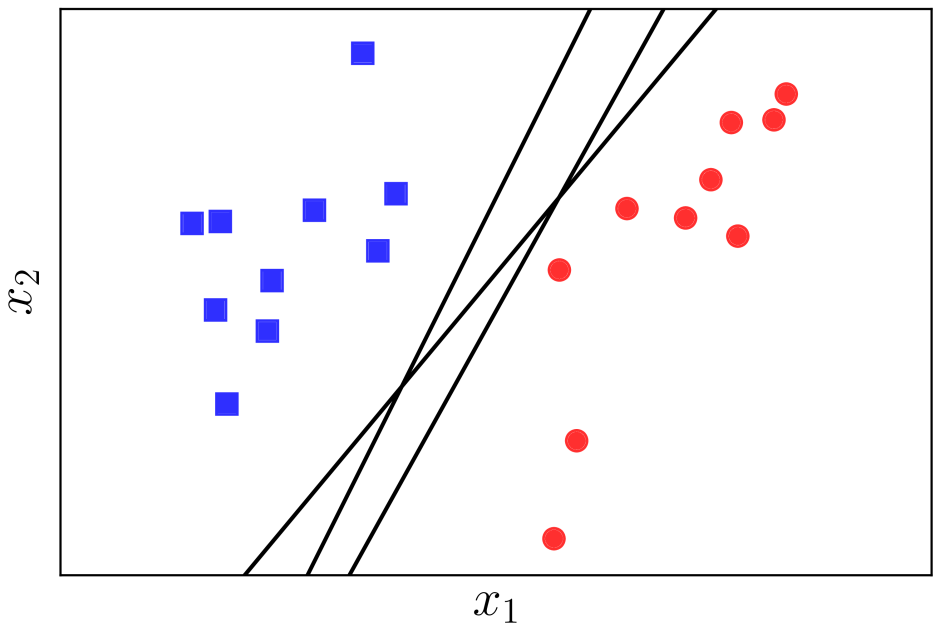
```
Predicting class of d5: B
Probability of d5 in each class: [[0.89548823 0.10451177]]
```

3 Xây dựng mô hình phân lớp SVM

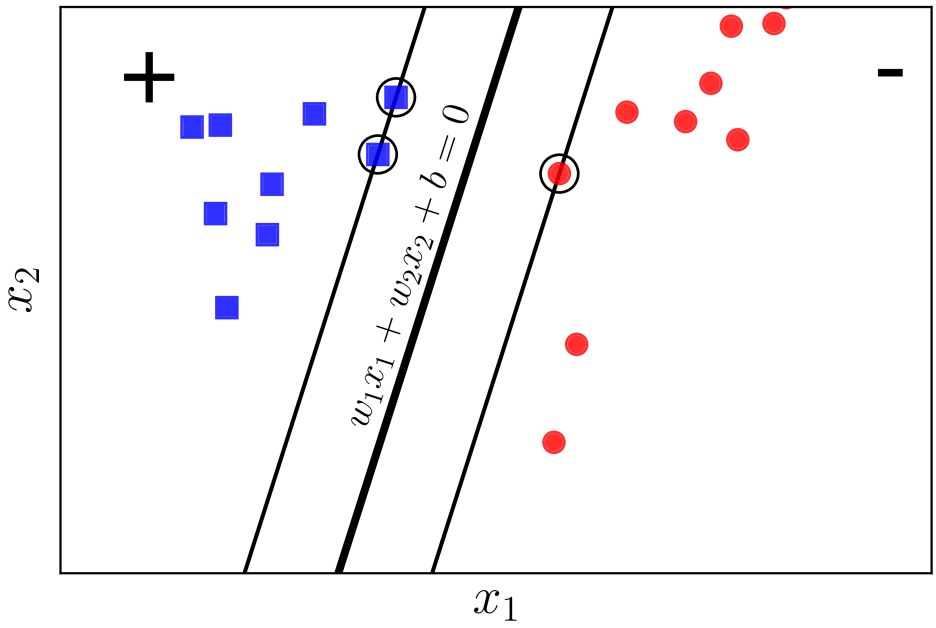
3.1 Cơ sở lý thuyết

SVM (Support Vector Machine) là một thuật toán học máy có giám sát được sử dụng rất phổ biến ngày nay trong các bài toán phân lớp (classification) hay hồi qui (Regression).

Ý tưởng của SVM là tìm một siêu phẳng (hyper lane) để phân tách các điểm dữ liệu. Siêu phẳng này sẽ chia không gian thành các miền khác nhau và mỗi miền sẽ chứa một loại giữ liệu.



Giả sử chúng ta phải phân loại tập dữ liệu các lớp dương (màu xanh) nhãn là 1 và các dữ liệu lớp âm (màu đỏ) nhãn là -1 (tập dữ liệu có thể phân tách tuyến tính). Siêu phẳng tối ưu nhất phải là siêu phẳng chia dữ liệu thành 2 classes có margin (khoảng cách từ siêu phẳng đến điểm dữ liệu gần nhất) là bằng nhau và lớn nhất có thể.



Giả sử rằng các cặp dữ liệu của training set là (x_1, y_1), (x_2, y_2), ..., (x_N, y_N) với vector $x_i \in R^d$ thể hiện đầu vào của một điểm dữ liệu và y_i là nhãn của điểm dữ liệu đó. d là số chiều của dữ liệu và N là số điểm dữ liệu. Giả sử rằng nhãn của mỗi điểm dữ liệu được xác định bởi $y_i = 1$ (class 1) hoặc $y_i =$ - 1 (class 2) giống như trong PLA.

Mặt phẳng phân chia được xác định bởi phương trình:

$$w^T x + b = w_1 x_1 + w_2 x_2 + b = 0 \quad (1)$$

Chúng ta cần đi tìm các hệ số w và b.

Với mặt phân chia như trên, margin được tính là khoảng cách gần nhất từ 1 điểm tới mặt đó (bất kể điểm nào trong hai classes):

$$margin = \min_n \frac{y_n(w^T x_n + b)}{||w||_2} \quad (2)$$

Bài toán tối ưu trong SVM chính là bài toán tìm **w** và b sao cho margin này đạt giá trị lớn nhất:

$$(w, b) = \arg\max_{w,b} \left(\min_n \frac{y_n(w^T x_n + b)}{||w||_2} \right) = \arg\max_{w,b} \left(\min_n \frac{1}{||w||_2} y_n(w^T x_n + b) \right) \quad (3)$$

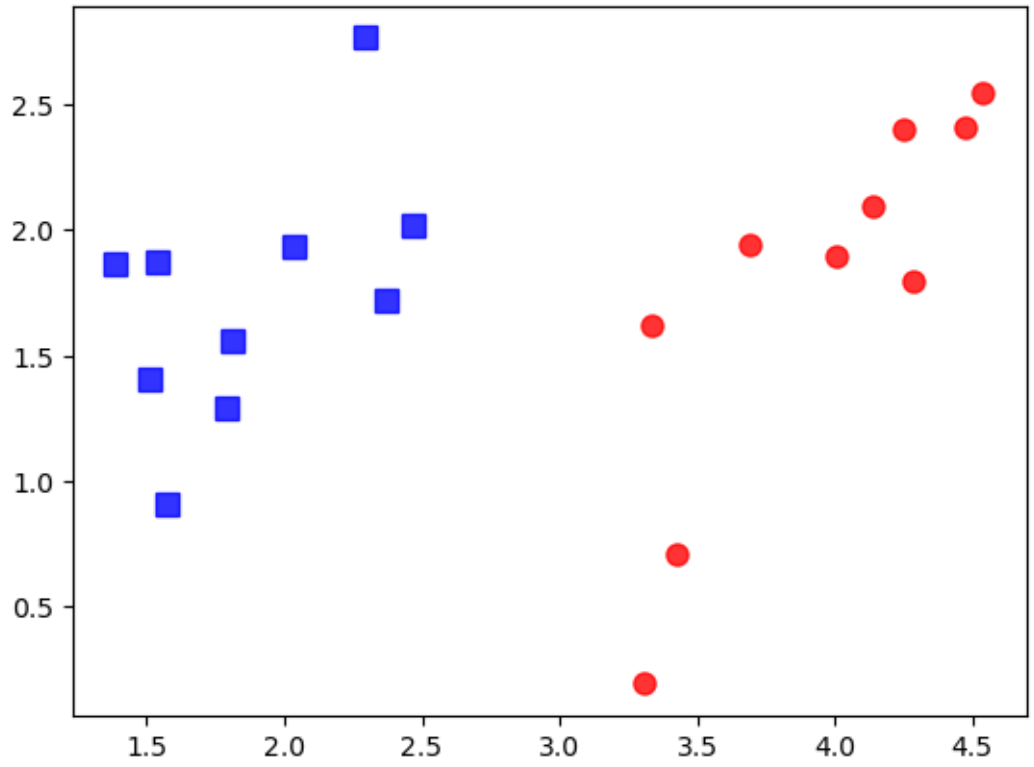
3.2 Ví dụ

[11]:
```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC

np.random.seed(22)

means = [[2, 2], [4, 2]]
cov = [[.3, .2], [.2, .3]]
N = 10
X0 = np.random.multivariate_normal(means[0], cov, N) # class 1
X1 = np.random.multivariate_normal(means[1], cov, N) # class -1
X = np.concatenate((X0.T, X1.T), axis = 1) # all data
y = np.concatenate((np.ones((1, N)), -1*np.ones((1, N))), axis = 1) # labels

# Vẽ các điểm dữ liệu
plt.plot(X0[:, 0], X0[:, 1], 'bs', markersize = 8, alpha = .8)
plt.plot(X1[:, 0], X1[:, 1], 'ro', markersize = 8, alpha = .8)
plt.show()
```



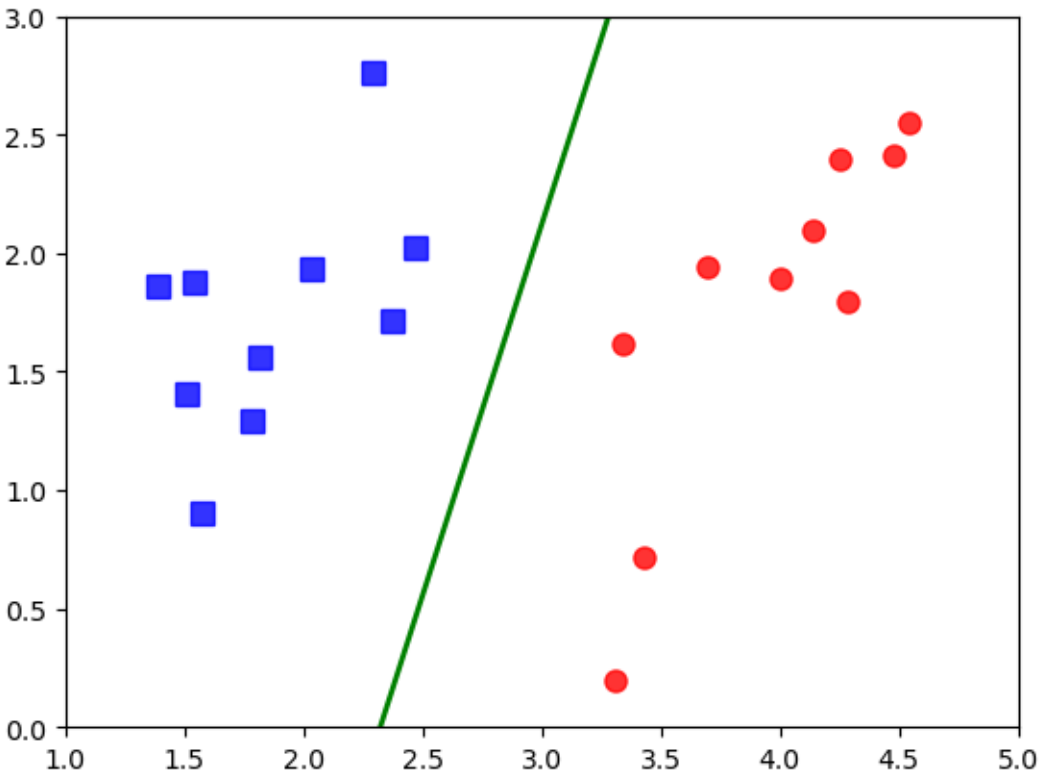
[12]:
```
y = y.reshape((2*N,))
X = X.T # each sample is one row
clf = SVC(kernel = 'linear', C = 1e5) # just a big number

clf.fit(X, y)

w = clf.coef_
b = clf.intercept_
print('w = ', w)
print('b = ', b)

# Vẽ minh họa các điểm dữ liệu và đường phân cách
x1 = np.arange(1, 5, 0.1)
y1 = -w[0, 0]/w[0, 1]*x1 - b/w[0, 1]
plt.axis([1, 5, 0, 3])
plt.plot(X0[:, 0], X0[:, 1], 'bs', markersize = 8, alpha = .8)
plt.plot(X1[:, 0], X1[:, 1], 'ro', markersize = 8, alpha = .8)
plt.plot(x1, y1, 'g', linewidth = 2)
plt.show()
```

```
w =  [[-2.00971102  0.64194082]]
b =  [4.66595309]
```



4 Xây dựng mô hình phân lớp với giải thuật giảm gradient

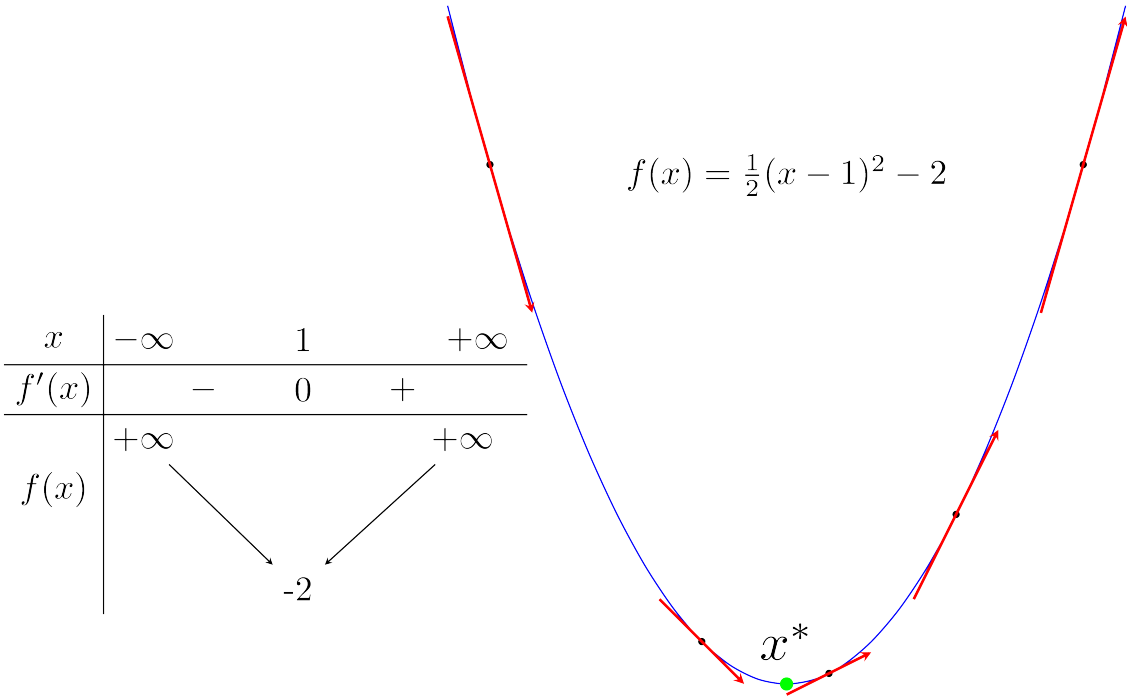
4.1 Cơ sở lý thuyết

Trong Machine Learning nói riêng và Toán Tối Ưu nói chung, chúng ta thường xuyên phải tìm giá trị nhỏ nhất (hoặc đôi khi là lớn nhất) của một hàm số nào đó. Nhìn chung, việc tìm global minimum của các hàm mất mát trong Machine Learning là rất phức tạp, thậm chí là bất khả thi. Thay vào đó, người ta thường cố gắng tìm các điểm local minimum, và ở một mức độ nào đó, coi đó là nghiệm cần tìm của bài toán.

Các điểm local minimum là nghiệm của phương trình đạo hàm bằng 0. Tuy nhiên, trong hầu hết các trường hợp, việc giải phương trình đạo hàm bằng 0 là bất khả thi. Nguyên nhân có thể đến từ sự phức tạp của dạng của đạo hàm, từ việc các điểm dữ liệu có số chiều lớn, hoặc từ việc có quá nhiều điểm dữ liệu.

Hướng tiếp cận phổ biến nhất là xuất phát từ một điểm mà chúng ta coi là gần với nghiệm của bài toán, sau đó dùng một phép toán lặp để tiến dần đến điểm cần tìm, tức đến khi đạo hàm gần với 0. Gradient Descent (viết gọn là GD) và các biến thể của nó là một trong những phương pháp được dùng nhiều nhất.

Giả sử hàm số f(x) có điểm local minimum x^*, x_t là điểm ta tìm được sau vòng lặp thứ t. Ta cần tìm một thuật toán để đưa x_t về càng gần x^* càng tốt.



- Nếu đạo hàm của hàm số tại x_t: $f'(x_t) > 0$ thì x_t nằm về phía bên phải so với x^*. Để điểm x_{t+1} tiến về x^*, ta cần di chuyển x_t về bên trái. Nói cách khác, ta cần di chuyển x_t theo hướng ngược với đạo hàm của hàm số tại x_t.

$$x_{t+1} = x_t + \Delta$$

trong đó Δ là một đại lượng ngược dấu với $f'(x_t)$.

- x_t càng xa x^* về phía bên phải thì $f'(x_t)$ càng lớn hơn 0 (và ngược lại). Vậy lượng di chuyển Δ tỉ lệ thuận với $f'(x_t)$.

Tổng quát, ta có công thức cập nhật:

$$x_{t+1} = x_t - \eta f'(x_t)$$

trong đó η là một hằng số dương, thường được gọi là learning rate.

Một số biến thể của Gradient Descent:

- Batch Gradient Descent: Cập nhật dựa trên toàn bộ dữ liệu.
- Stochastic Gradient Descent (SGD): Thay vì cập nhật dựa trên toàn bộ dữ liệu, ta chỉ cập nhật dựa trên một điểm dữ liệu ngẫu nhiên.
- Mini-batch Gradient Descent: Cập nhật dựa trên một số lượng nhỏ các điểm dữ liệu.

4.2 Implement a Stochastic Gradient Descent Classifier in Python

[14]:
```
# importing Libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns

# Load the Iris dataset
data = load_iris()
X, y = data.data, data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
```

[15]:
```
# Create an SGD Classifier
clf = SGDClassifier(loss='log_loss', alpha=0.01,max_iter=1000, random_state=42)

# Train the classifier
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)
```

```python
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')

# Plot the confusion matrix using Seaborn
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", cbar=False,
                        xticklabels=data.target_names, yticklabels=data.
 ↪target_names)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.9555555555555556
```