# 1 (8 points) Conceptual Questions

**1.** Choose **all** the valid answers to the description about **linear regression** and **logistic regression** from the options below:

A. Linear regression is an unsupervised learning problem; logistic regression is a supervised learning problem.

B. Linear regression deals with the prediction of continuous values; logistic regression deals wit

C. We cannot use gradient descent to solve linear regression. Instead, we can only use the closed-form solution to tackle the linear regression problem.

D. Linear regression is a convex optimization problem whereas logistic regression is not.

**2.** Choose **all** the valid answers to the description about **gradient descent** from the options below:

A. The global minimum is guaranteed to be reached by using gradient descent.

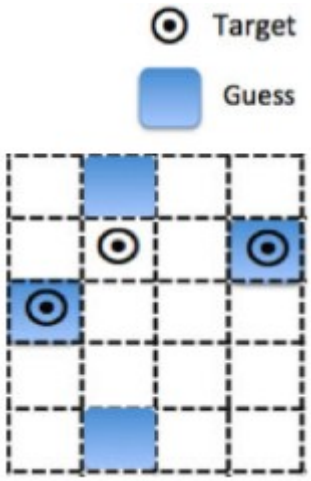
B. Every gradient descent iteration can always decrease the value of loss function even when the gradient of the loss function is zero.

C. When the learning rate is very large, it is possible that some iterations of gradient descent m

D. With different initial weights, it is possible for the gradient descent algorithm to obtain to d

# 2 (12 points) Error Metrics

The grid below shows 5×4 = 20 possible locations for targets to appear. We make guesses at the locations where targets are located, and mark those cells in blue. For the rest of the locations, we guess that those locations are non-targets, which are marked in white. The actual locations for the target are marked with a circle in cells, while the actual locations for non-target are marked with empty cells. Evaluate the following metrics for your guesses.




**1.** Compute the Recall using the following formula:

$$\text{Recall} = \frac{\text{number of true positives (correct guesses)}}{\text{number of actual targets}}$$

$$\boxed{\text{Recall} = \frac{2}{3}}$$

**2.** Compute the Precision using the following formula:

$$\text{Precision} = \frac{\text{number of true positives (correct guesses)}}{\text{number of guesses}}$$

$$\boxed{\text{Precision} = \frac{2}{4} = \frac{1}{2}}$$

**3.** Compute the F1 metric on this data. F1 is a way to measure classification accuracy. It's the harmonic mean of the precision and recall. F1 value is calculated using the following formula:

$$\text{F1} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{F1} = \frac{2 * \frac{1}{2} * \frac{2}{3}}{\frac{1}{2} + \frac{2}{3}}$$

$$\boxed{\text{F1} = \frac{\frac{2}{3}}{\frac{7}{6}} = \frac{12}{21} = \frac{4}{7}}$$

# 3 (10 points) Logistic Regression Inference

We have a logistic regression classifier for a 2-dimensional feature vector $x = (x_1, x_2)$:

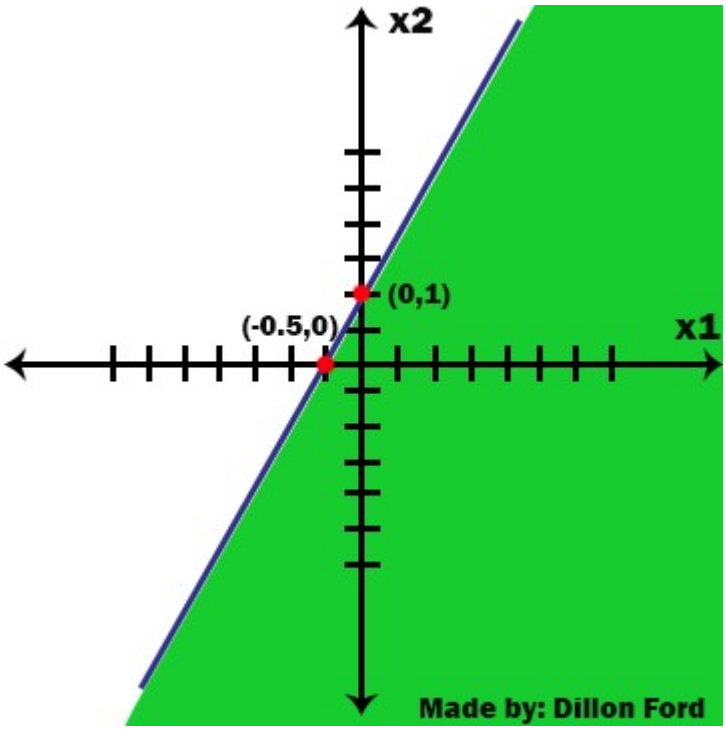
$$p(y = +1|x) = \frac{1}{1 + e^{-(2x_1 - x_2 + 1)}}$$

$$f(x) = \begin{cases} 1, \ p(y = +1|x) \geq 0.5 \\ -1, \ \text{otherwise} \end{cases}$$

where $f(x) \ \varepsilon \ \{-1, 1\}$ is the predicted label. Please solve the following problems

**1.** Draw the decision boundary of the logistic regression classifier and shade the region where the classifier predicts 1. Make sure you have marked the $x_1$ and $x_2$ axes and the intercepts on those axes.

$$\begin{gathered} \frac{1}{1 + e^{-(2x_1 - x_2 + 1)}} \geq 0.5 \\ 1 \geq 0.5(1 + e^{-(2x_1 - x_2 + 1)}) \\ \frac{1}{0.5} \geq 1 + e^{-(2x_1 - x_2 + 1)} \\ 2 \geq 1 + e^{-(2x_1 - x_2 + 1)} \\ 1 \geq e^{-(2x_1 - x_2 + 1)} \\ ln(1) \geq -(2x_1 - x_2 - 1) \\ 0 \geq -2x_1 + x_2 - 1 \\ -x_2 \geq -2x_1 - 1 \\ x_2 \leq 2x_1 + 1 \end{gathered}$$




**2.** There are two new data points $x_1 = (0.5, 3)$ and $x_2 = (0.5, -2)$. Use the logistic regression defined above to:

a. Calculate the probability that each data point belongs to the positive class. You may use a calculator or computer to get the numeric answer.

$$x_1 = (0.5, 3)$$

$$p(y = +1|x_1) = \frac{1}{1 + e^{-(2(0.5)-(3)+1)}}$$

$$p(y = +1|x_1) = \frac{1}{1 + e^{-(1-2)}}$$

$$p(y = +1|x_1) = \frac{1}{1 + e}$$

$$\boxed{p(y = +1|x_1) \approx 0.27}$$

$$x_2 = (0.5, -2)$$
$$p(y = +1|x_2) = \frac{1}{1 + e^{-(2(0.5)-(-2)+1)}}$$
$$p(y = +1|x_2) = \frac{1}{1 + e^{-(1+3)}}$$
$$p(y = +1|x_2) = \frac{1}{1 + e^{-}4}$$
$$\boxed{p(y = +1|x_2) \approx 0.98}$$

b. Predict the labels for the new data points.

$$x_1 = (0.5, 3)$$
$$\text{given that,}$$
$$p(y = +1|x_1) \approx 0.27 \ngeq 0.5$$
$$\boxed{\text{label for } x_1 \text{ is } -1}$$

$$x_2 = (0.5, -2)$$
$$\text{given that,}$$
$$p(y = +1|x_2) \approx 0.98 \geq 0.5$$
$$\boxed{\text{label for } x_2 \text{ is } 1}$$

# 4 (40 points) Logistic Regression

Assume in a binary classification problem, we need to predict a binary label $y \varepsilon \{-1, 1\}$ for a feature vector $x = [x_0, x_1]^T$. In logistic regression, we can reformulate the binary classification problem in a probabilistic framework: We aim to model the distribution of classes given the input feature vector $x$. Specifically, we can express the conditional probability $p(y|\boldsymbol{x})$ parameterized by $(\boldsymbol{w}, b)$ using logistic function as:

$$p(y|x) = \frac{1}{1 + e^{-y(w \cdot x + b)}}$$

where $w = [w_0, w_1]^T$ is the weight vector, and b is the bias scalar. Given a training data set $S_{\text{training}} = \{(x_i, y_i)\},\ i = 1, \ldots, n\}$, we wish to optimize the negative log-likelihood loss $L(w, b)$:

$$\mathcal{L}(w,b) = -\sum_{i=1}^{n} \ln p(y_i|x_i)$$

and find the optimal weight vector w and bias b to build the logistic regression model:

$$w^*, b^* = \text{argmin}_{w,b}\mathcal{L}(w,b)$$

- In this problem, we attempt to obtain the optimal parameters $w^*$ and $b^*$ by using a standard gradient descent algorithm. Assume $p_i = p(y_i|x_i)$, the gradient for $w$ and the gradient for $b$ are shown as following:

$$\frac{\partial \mathcal{L}(w,b)}{\partial w} = -\sum_{i=1}^{n}(1-p_i)y_i x_i, \;\; \frac{\partial \mathcal{L}(w,b)}{\partial b} = -\sum_{i=1}^{n}(1-p_i)y_i$$

In reality, we typically tackle this problem in a matrix form: First, we represent data points as matrices $X = [x_1, x_2, \ldots, x_n]^T$ and $Y = [y_1, y_2, \ldots, y_n]^T$. Thus, the negative log-likelihood loss $\mathcal{L}(w,b)$ can be formulated as:

$$P = \text{sigmoid}(Y \circ (Xw + b1)), \;\; \mathcal{L}(w,b) = -1^T \ln P$$

where $1 = [1, 1, \ldots, 1]^T \; \varepsilon \; R^n$ is a n-dimensional column vector, ln(·) is an element-wise natural logarithm function, sigmoid $(z) = \frac{1}{1+e^{-z}}$ is an element-wise sigmoid function, and "∘" is an element-wise product operator. Similarly, we can have the gradient for $w$ and $b$ in the matrix form:

$$\frac{\partial \mathcal{L}(w,b)}{\partial w} = -X^T((1-P) \circ Y), \;\; \frac{\partial \mathcal{L}(w,b)}{\partial b} = -1^T((1-P) \circ Y).$$

- After obtaining the logistic regression model in Eq. 1 with the optimal $w^*$, $b^*$ from gradient descent, we can use the following decision rule to predict the label $f(x; w^*, b^*) \; \varepsilon \; \{-1, 1\}$ of the feature vector $x$:

$$f(x; w^*, b^*) = \begin{cases} 1, \; p(y = +1|x) \geq 0.5 \\ -1, \; \text{otherwise} \end{cases}$$

Besides, the error $e_{\text{training}}$ on the training set $S_{\text{training}} = \{(x_i, y_i)\}$ is defined as:

$$e_{\text{training}} = \frac{1}{n}\sum_{i=1}^{n} 1(y_i \neq f(x_i; w^*, b^*))$$

and we can define the test error $e_{\text{test}}$ on the test set $S_{\text{test}}$ in the same way.

Please download the notebook **logistic-regression.ipynb** from the course Canvas and fill in the missing blanks. Follow the instructions in the skeleton code and report:

1. Your code.
2. Equation of decision boundary corresponding to the optimal $w^*$ and $b^*$.
3. Plot of training set along with decision boundary.
4. Plot of test set along with decision boundary.
5. Training error and test error.
6. Training loss curve.

## Logistic Regression

In this notebook you will use your knowledge of the mathematics of logistic regression to write a program to solve for the logistic parameters given a training dataset.

The necessary math is in question #4, homework #4.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
```

In [2]:
```
# Iris dataset.
iris = datasets.load_iris()      # Load Iris dataset.

X = iris.data                    # The shape of X is (150, 4), which mea
ns
                                 # there are 150 data points, each data
point
                                 # has 4 features.

# Here for convenience, we divide the 3 kinds of flowers into 2 groups:
#     Y = 0 (or False):  Setosa (original value 0) / Versicolor (origin
al value 1)
#     Y = 1 (or True):   Virginica (original value 2)

# Thus we use (iris.target > 1.5) to divide the targets into 2 groups.
# This line of code will assign:
#    Y[i] = True  (which is equivalent to 1) if iris.target[k]  > 1.5
(Virginica)
#    Y[i] = False (which is equivalent to 0) if iris.target[k] <= 1.5
(Setosa / Versicolor)

Y = (iris.target > 1.5).reshape(-1,1).astype(np.float) # The shape of Y
is (150, 1), which means
                                 # there are 150 data points, each data
point
                                 # has 1 target value.
Y[Y==0] = -1

X_and_Y = np.hstack((X, Y))      # Stack them together for shuffling.
np.random.seed(1)                # Set the random seed.
np.random.shuffle(X_and_Y)       # Shuffle the data points in X_and_Y ar
ray

print(X.shape)
print(Y.shape)
print(X_and_Y[0])                # The result should be always: [ 5.8
4.   1.2  0.2  -1. ]
```

```
(150, 4)
(150, 1)
[ 5.8  4.   1.2  0.2 -1. ]
```

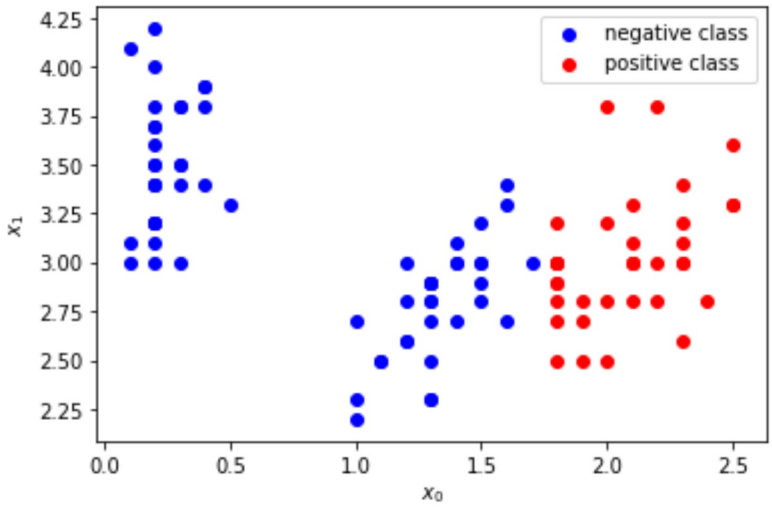
In [3]:
```
# Divide the data points into training set and test set.
X_shuffled = X_and_Y[:,:4]
Y_shuffled = X_and_Y[:,4]


X_train = X_shuffled[:100][:,[3,1]] # Shape: (100,2)
X_train = np.delete(X_train, 42, axis=0) # Remove a point for separabil
ity.
Y_train = Y_shuffled[:100]          # Shape: (100,)
Y_train = np.delete(Y_train, 42, axis=0) # Remove a point for separabil
ity.
X_test = X_shuffled[100:][:,[3,1]]  # Shape: (50,2)
Y_test = Y_shuffled[100:]           # Shape: (50,)
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)
```

```
(99, 2)
(99,)
(50, 2)
(50,)
```

## Visualization

In [4]:
```
def vis(X, Y, W=None, b=None):
    indices_neg1 = (Y == -1).nonzero()[0]
    indices_pos1 = (Y == 1).nonzero()[0]
    plt.scatter(X[:,0][indices_neg1], X[:,1][indices_neg1],
                c='blue', label='negative class')
    plt.scatter(X[:,0][indices_pos1], X[:,1][indices_pos1],
                c='red', label='positive class')
    plt.legend()
    plt.xlabel('$x_0$')
    plt.ylabel('$x_1$')

    if W is not None:
        # w0x0+w1x1+b=0 => x1=-w0x0/w1-b/w1
        w0 = W[0]
        w1 = W[1]
        temp = -w1*np.array([X[:,1].min(), X[:,1].max()])/w0-b/w0
        x0_min = max(temp.min(), X[:,0].min())
        x0_max = min(temp.max(), X[:,1].max())
        x0 = np.linspace(x0_min,x0_max,100)
        x1 = -w0*x0/w1-b/w1
        plt.plot(x0,x1,color='black')

    plt.show()
```

In [5]:
```
# Visualize training set.
vis(X_train, Y_train)
```




## Logistic Regression Using Gradient Descent

In this problem, we would like to use the gradient descent to calculate the parameters $\mathbf{w}, b$ for a logistic regression model. If we have the loss function $\mathcal{L}(\mathbf{w}, b)$, then a typical gradient descent algorithm contains the following steps:

**Step 1**. Initialize the parameters $\mathbf{w}$, $b$.

for i = 1 to #iterations:

- **Step 2**. Compute the partial derivatives $\frac{\partial \mathcal{L}(\mathbf{w},b)}{\partial \mathbf{w}}$, $\frac{\partial \mathcal{L}(\mathbf{w},b)}{\partial b}$.
- **Step 3**. Update the parameters

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial \mathcal{L}(\mathbf{w}, b)}{\partial \mathbf{w}}, \qquad b \leftarrow b - \eta \frac{\partial \mathcal{L}(\mathbf{w}, b)}{\partial b}$$

  where $\eta$ is the learning rate.

Note that in the code, we use `W` and `b` to represent the weight vector $\mathbf{w}$ and bias scalar $b$.

In [6]:
```
# Sigmoid function: sigmoid(z) = 1/(1 + e^(-z))
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

In [7]:
```
# Judge function: 1(a != b).
def judge(a, b):

    if a!=b:
        return 1
    return 0

# Logistic regression classifier.
def f_logistic(x, W, b):
    # x should be a 2-dimensional vector,
    # W should be a 2-dimensional vector,
    # b should be a scalar.
    # you should return a scalar which is -1 or 1.
    # Hint: use sigmoid() function.

    z = np.dot(W,x)+b
    if sigmoid(z) >= 0.5:
        return 1
    return -1


# Calculate error given feature vectors X and labels Y.
def calc_error(X, Y, W, b):
    sum_values = []
    for (xi, yi) in zip(X, Y):
        # Hint: Use judge() and f_logistic()
        sum_values.append(judge(yi, f_logistic(xi,W,b)))

    return np.mean(sum_values)
```

In [8]:
```
# Gradient of L(W, b) with respect to W and b.
def grad_L_W_b(X, Y, W, b):

    P = sigmoid(np.multiply(Y, np.dot(X,W)+b))
    grad_W = np.dot(-X.T, np.multiply((1-P),Y))
    grad_b = -np.dot(np.ones(len(X)).T, np.multiply((1-P),Y))

    return grad_W, grad_b
```

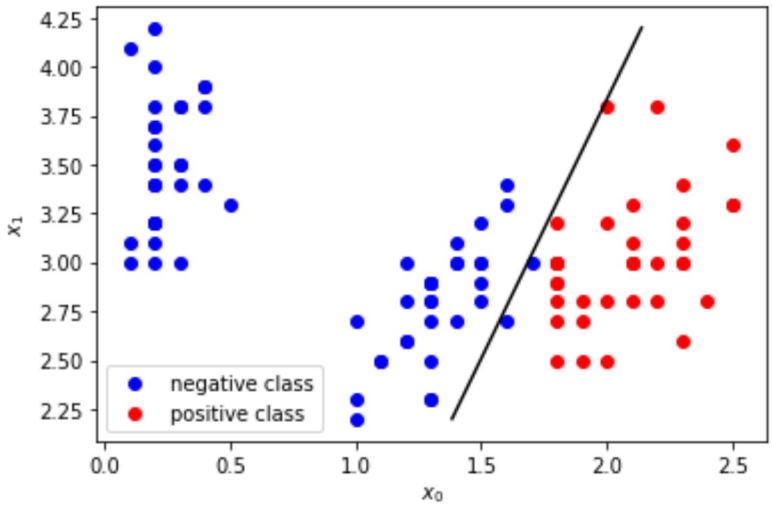
In [9]:
```
# Loss L(W, b).
def L_W_b(X, Y, W, b):
    # You should return a scalar.
    P = sigmoid(np.multiply(Y, np.dot(X,W)+b))
    loss = np.dot(-np.ones(len(X)).T, np.log(P))
    return loss
```

In [10]:
```
# Some settings.
learning_rate = 0.001
iterations    = 10000
losses = []

# Gradient descent algorithm for logistic regression.
# Step 1. Initialize the parameters W, b.
W = np.zeros(2)
b = 0

for i in range(iterations):
    # Step 2. Compute the partial derivatives.
    grad_W, grad_b = grad_L_W_b(X_train, Y_train, W, b)
    # Step 3. Update the parameters.
    # Update W.
    W = W - learning_rate*(grad_W)
    # Update b.
    b = b - learning_rate*(grad_b)
    # Track the training losses.
    losses.append(L_W_b(X_train, Y_train, W, b))
```

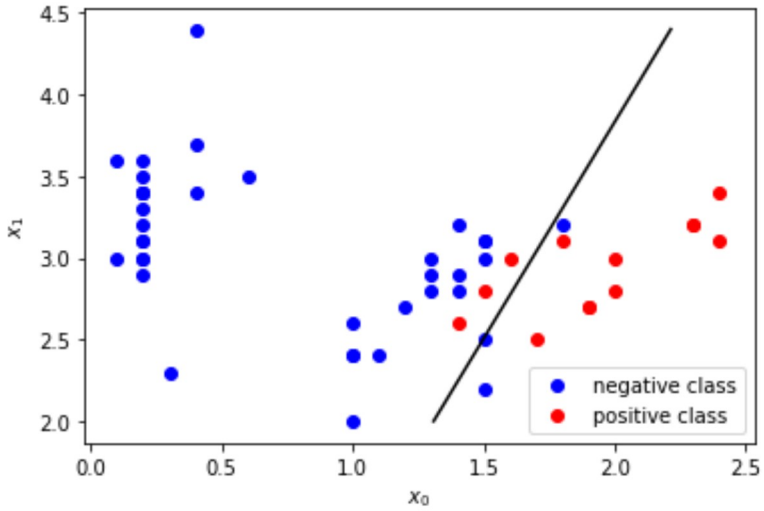
## Visualize the results

In [11]:
```
# Show decision boundary, training error and test error.
print('Decision boundary: {:.3f}x0+{:.3f}x1+{:.3f}=0'.format(W[0],W[1],
b))
vis(X_train, Y_train, W, b)
print('Training error: {}'.format(calc_error(X_train, Y_train, W, b)))
vis(X_test, Y_test, W, b)
print('Test error: {}'.format(calc_error(X_test, Y_test, W, b)))
```

```
Decision boundary: 11.666x0+-4.401x1+-6.462=0
```

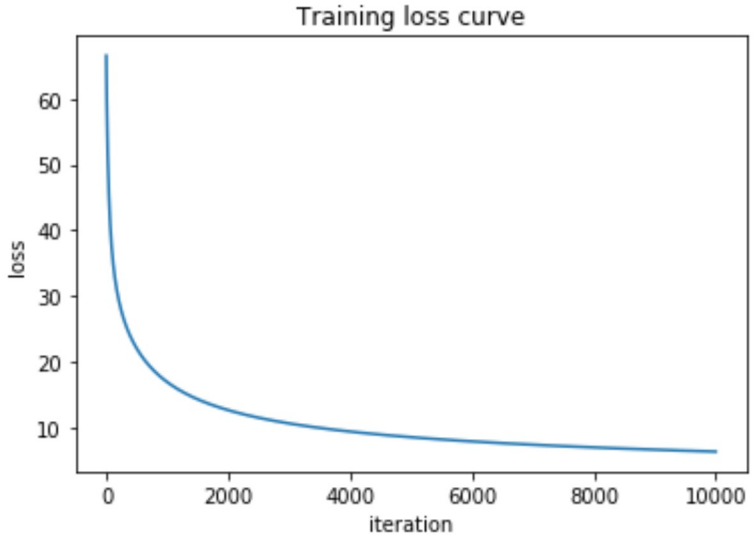



```
Training error: 0.020202020202020204
```




```
Test error: 0.12
```

In [12]:
```
# Plot training loss curve.
plt.title('Training loss curve')
plt.xlabel('iteration')
plt.ylabel('loss')
plt.plot(losses)
plt.show()
```




## 5 (30 points) Perceptron

In this section, we will apply perceptron learning algorithm to solve the same binary classification problem as logistic regression above: We need to predict a binary label $y \varepsilon \{-1, 1\}$ for a feature vector $x = [x_0, x_1]^T$. The decision rule of the perceptron model is defined as:

$$f(x; w, b) = \begin{cases} 1, \text{ if } w^T x + b \geq 0 \\ -1, \text{ otherwise} \end{cases}$$

where $w = [w0, w1]^T$ is the weight vector, and $b$ is the bias scalar. Given a training data set $S_{\text{training}} = \{(x_i, y_i)\},\ i = 1, \ldots, n\}$, we define the training error $e_{\text{training}}$ as:

$$e_{\text{training}} = \frac{1}{n} \sum_{i=1}^{n} 1(y_i \neq f(x_i; w, b))$$

and the test error $e_{\text{test}}$ on the test set $S$ test can be defined in the same way. In the perceptron algorithm, we aim to directly minimize the training error $e_{\text{training}}$ in order to obtain the optimal parameters $w^*,\ b^*$. If we represent data points in training set $S_{\text{training}}$ as matrices $X = [x_1, x_2, \ldots, x_n]^T$ and $Y = [y_1, y_2, \ldots, y_n]^T$, the perceptron algorithm is shown as below:

**Algorithm 1** Perceptron Learning Algorithm

**Data:** Training set $S_{\text{training}}$, which contains feature vectors $X$ and labels $Y$;
Initialize parameters $\mathbf{w}$ and $b$; pick a constant $\lambda \in (0, 1]$, which is similar to the step size in the standard gradient descent algorithm ($\lambda = 1$ by default).
**while** *not every data point is correctly classified* **do**
  **for** *each feature vector* $\mathbf{x}_i$ *and its label* $y_i$ *in the training set* $S_{training}$ **do**
    compute the model prediction $f(\mathbf{x}_i; \mathbf{w}, b)$;
    **if** $y_i = f(\mathbf{x}_i; \mathbf{w}, b)$ **then**
      **continue;**
    **else**
      update the parameters $\mathbf{w}$ and $\mathbf{b}$:
      $\mathbf{w} \Leftarrow \mathbf{w} + \lambda(y_i - f(\mathbf{x}_i; \mathbf{w}, b))\mathbf{x}_i$
      $\mathbf{b} \Leftarrow \mathbf{b} + \lambda(y_i - f(\mathbf{x}_i; \mathbf{w}, b))$
    **end**
  **end**
**end**

Please download the notebook **perceptron.ipynb** from the course website and fill inthe missing blanks. Follow the instructions in the skeleton code and report:

1. Your code.
2. Equation of decision boundary corresponding to the optimal $w^*$ and $b^*$.
3. Plot of training set along with decision boundary.
4. Plot of test set along with decision boundary.
5. Training error and test error.
6. Training error curve.

## Perceptron

In this notebook you will use your knowledge of the mathematics of perceptrons to write a program to solve for the weights given a training dataset.

The necessary math is in question #5, homework #4.

In [13]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
```

## Load the modified Iris dataset

In [14]:
```
# Iris dataset.
iris = datasets.load_iris()      # Load Iris dataset.

X = iris.data                    # The shape of X is (150, 4), which mea
ns
                                 # there are 150 data points, each data
point
                                 # has 4 features.

# Here for convenience, we divide the 3 kinds of flowers into 2 groups:
#     Y = 0 (or False):  Setosa (original value 0) / Versicolor (origin
al value 1)
#     Y = 1 (or True):   Virginica (original value 2)

# Thus we use (iris.target > 1.5) to divide the targets into 2 groups.
# This line of code will assign:
#    Y[i] = True  (which is equivalent to 1) if iris.target[k]  > 1.5
(Virginica)
#    Y[i] = False (which is equivalent to 0) if iris.target[k] <= 1.5
(Setosa / Versicolor)

Y = (iris.target > 1.5).reshape(-1,1).astype(np.float) # The shape of Y
is (150, 1), which means
                                 # there are 150 data points, each data
point
                                 # has 1 target value.
Y[Y==0] = -1

X_and_Y = np.hstack((X, Y))      # Stack them together for shuffling.
np.random.seed(1)                # Set the random seed.
np.random.shuffle(X_and_Y)       # Shuffle the data points in X_and_Y ar
ray

print(X.shape)
print(Y.shape)
print(X_and_Y[0])                # The result should be always: [ 5.8
4.   1.2  0.2  -1. ]
```

```
(150, 4)
(150, 1)
[ 5.8  4.   1.2  0.2 -1. ]
```

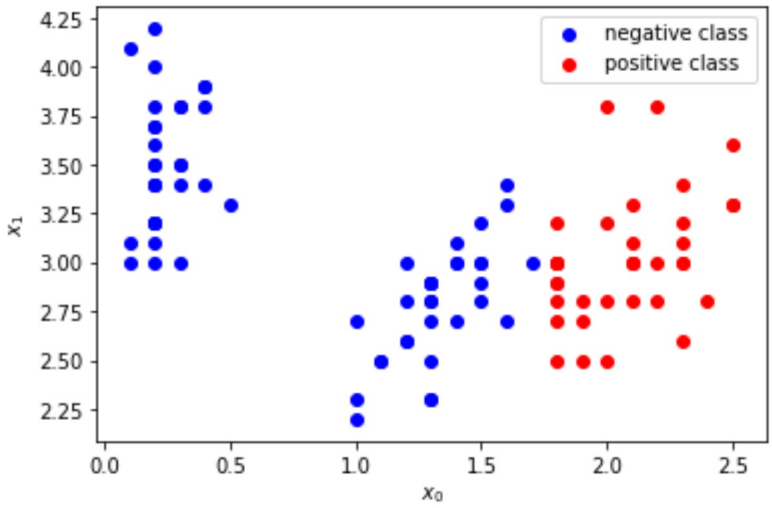
In [15]:
```
# Divide the data points into training set and test set.
X_shuffled = X_and_Y[:,:4]
Y_shuffled = X_and_Y[:,4]


X_train = X_shuffled[:100][:,[3,1]] # Shape: (100,2)
X_train = np.delete(X_train, 42, axis=0) # Remove a point for separabil
ity.
Y_train = Y_shuffled[:100]          # Shape: (100,)
Y_train = np.delete(Y_train, 42, axis=0) # Remove a point for separabil
ity.
X_test = X_shuffled[100:][:,[3,1]]  # Shape: (50,2)
Y_test = Y_shuffled[100:]           # Shape: (50,)
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)
```

```
(99, 2)
(99,)
(50, 2)
(50,)
```

## Visualization

In [16]:
```
def vis(X, Y, W=None, b=None):
    indices_neg1 = (Y == -1).nonzero()[0]
    indices_pos1 = (Y == 1).nonzero()[0]
    plt.scatter(X[:,0][indices_neg1], X[:,1][indices_neg1],
                c='blue', label='negative class')
    plt.scatter(X[:,0][indices_pos1], X[:,1][indices_pos1],
                c='red', label='positive class')
    plt.legend()
    plt.xlabel('$x_0$')
    plt.ylabel('$x_1$')

    if W is not None:
        # w0x0+w1x1+b=0 => x1=-w0x0/w1-b/w1
        w0 = W[0]
        w1 = W[1]
        temp = -w1*np.array([X[:,1].min(), X[:,1].max()])/w0-b/w0
        x0_min = max(temp.min(), X[:,0].min())
        x0_max = min(temp.max(), X[:,1].max())
        x0 = np.linspace(x0_min,x0_max,100)
        x1 = -w0*x0/w1-b/w1
        plt.plot(x0,x1,color='black')

    plt.show()
```

In [17]:
```
# Visualize training set.
vis(X_train, Y_train)
```




## Perceptron Algorithm

In this problem, we would like to train a perceptron model for the classification task on a modified Iris dataset. The training procedure of the perceptron model is shown in the algorithm below:

**Algorithm 1** Perceptron Learning Algorithm

**Data:** Training set $S_{\text{training}}$, which contains feature vectors $X$ and labels $Y$;
Initialize parameters $\mathbf{w}$ and $b$; pick a constant $\lambda \in (0, 1]$, which is similar to the step size in the standard gradient descent algorithm ($\lambda = 1$ by default).
**while** *not every data point is correctly classified* **do**
  **for** *each feature vector* $\mathbf{x}_i$ *and its label* $y_i$ *in the training set* $S_{\text{training}}$ **do**
    compute the model prediction $f(\mathbf{x}_i; \mathbf{w}, b)$;
    **if** $y_i = f(\mathbf{x}_i; \mathbf{w}, b)$ **then**
      **continue;**
    **else**
      update the parameters $\mathbf{w}$ and $\mathbf{b}$:
      $\mathbf{w} \Leftarrow \mathbf{w} + \lambda(y_i - f(\mathbf{x}_i; \mathbf{w}, b))\mathbf{x}_i$
      $\mathbf{b} \Leftarrow \mathbf{b} + \lambda(y_i - f(\mathbf{x}_i; \mathbf{w}, b))$
    **end**
  **end**
**end**

Note that in the code, we use `X_train` and `Y_train` to represent the feature vector $X$ and labels $Y$ in training set $S_{\text{training}}$. Besides, we use `W` and `b` to represent the weight vector $\mathbf{w}$ and bias scalar $b$.

Please fill the blanks of the skeleton code below to complete the perceptron training procedure.

**Hint**: For the implementation of some functions, you may refer to HW2 Q5.

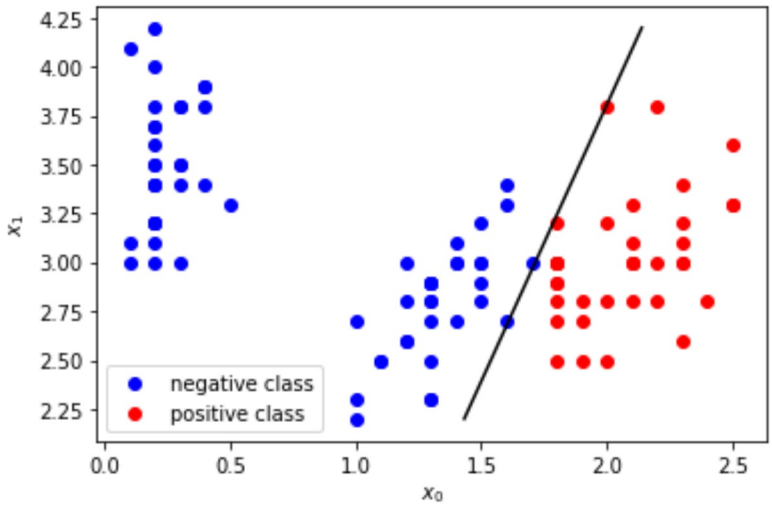
In [18]:
```
# Judge function: 1(a != b).
def judge(a, b):
    ####### To be filled #######
    if a!=b:
        return 1
    return 0

# Perceptron classifier.
def f_perceptron(x, W, b):
    # x should be a 2-dimensional vector,
    # W should be a 2-dimensional vector,
    # b should be a scalar.
    # you should return a scalar which is -1 or 1

    if np.dot(W.T,x)+b >= 0:
        return 1
    return -1

# Calculate error given feature vectors X and labels Y.
def calc_error(X, Y, W, b):
    ############ To be filled. ############

    calc_values = []

    for (xi, yi) in zip(X, Y):
        # Hint: Use judge() and f_perceptron()

        xi = f_perceptron(xi,W,b)
        calc_values.append(judge(yi, xi))


    return np.mean(calc_values)
```

In [19]:
```
# Some settings.
errors = []           # Error history.
lam    = 1            # Lambda which controls the step size.

# Initialization.
W      = np.zeros(2)  # Weight.
b      = 0.0          # Bias.

# Perceptron learning algorithm.
while calc_error(X_train, Y_train, W, b) > 0:
    for xi, yi in zip(X_train, Y_train):   # Iterate over all data poin
ts.
        pred = f_perceptron(xi,W,b)        # Compute the model predicti
on.
        if yi == pred:                     # Compare prediction and lab
el.
            continue                       # - If correct, continue.
        else:
            W = W + lam*(yi - pred)*xi     # - If not, update weight an
d bias.
            b = b + lam*(yi - pred)

    # Track training errors.
    errors.append(calc_error(X_train, Y_train, W, b))
```

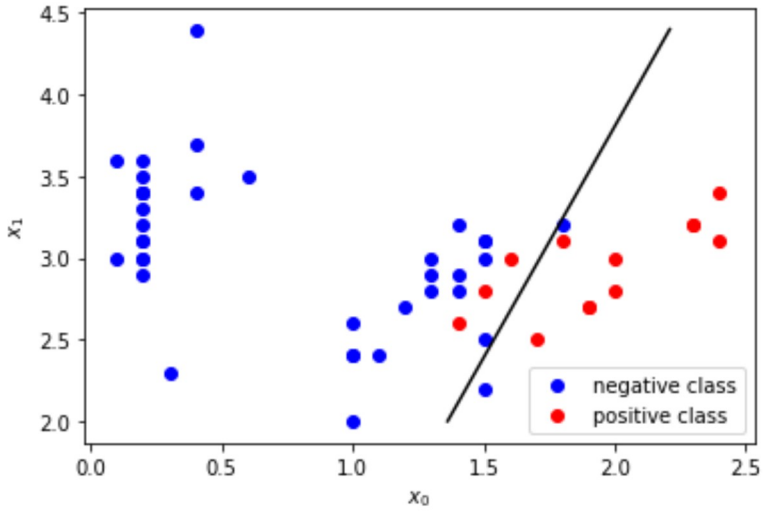
**Visualize the results**

In [20]:
```
# Show decision boundary, training error and test error.
print('Decision boundary: {:.3f}x0+{:.3f}x1+{:.3f}=0'.format(W[0],W[1],
b))
vis(X_train, Y_train, W, b)
print('Training error: {}'.format(calc_error(X_train, Y_train, W, b)))
vis(X_test, Y_test, W, b)
print('Test error: {}'.format(calc_error(X_test, Y_test, W, b)))
```

```
Decision boundary: 70.200x0+-24.800x1+-46.000=0
```




```
Training error: 0.0
```




```
Test error: 0.1
```

In [21]:
```
# Plot training error curve.
plt.title('Training error curve')
plt.plot(errors)
plt.xlabel('iteration')
plt.ylabel('error')
plt.show()
```