In [ ]:

```
start.bat
```

Iris Dataset

In [1]:

```python
# import some library we are gonna use
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sklearn
import sklearn.preprocessing as pre
```

In [2]:

```python
# Read the data from the iris.data file and show the first five item in the dataset
# the iris dataset have four features: sepal_len, sepal_wid, petal_len and petal_wid
# the last column is class which is the thing we want
df = pd.read_csv('data/iris/iris.data')
df.head()
```

Out[2]:

	sepal_len	sepal_wid	petal_len	petal_wid	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [3]:

```python
# We can do slicing in pandas DataFrame using the iloc method as in NumPy Array
X, y = df.iloc[:, :-1], df.iloc[:, -1]
```

In [4]:

```python
# Transform the DataFrame to numpy array
X, y = np.array(X), np.array(y)
```

In [5]:

```
# There is three classes in the iris dataset
set(y)
```

Out[5]:

```
{'Iris-setosa', 'Iris-versicolor', 'Iris-virginica'}
```

In [6]:

```
y[:20]
```

Out[6]:

```
array(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
       'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
       'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
       'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
       'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa'],
      dtype=object)
```

In [7]:

```
# Since the y(the label) need to a numerical for numpy to process
# here we map the classes name to a interger represent to classes
for idx, class_name in enumerate(sorted(list(set(y)))):
    y[y == class_name] = idx
```

In [8]:

```
y
```

Out[8]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], dtype=object)
```

In [9]:

```
y = pre.LabelEncoder().fit_transform(df.iloc[:, -1])
```

In [10]:

```
y
```

Out[10]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

In [11]:

```
X.shape, y.shape
```

Out[11]:

```
((150, 4), (150,))
```

In [12]:

```
# We can also use sklearn to help us load in the iris dataset
from sklearn.datasets import load_iris
X, y = load_iris(return_X_y=True)
```

In [13]:

```
X.shape, y.shape
```

Out[13]:

```
((150, 4), (150,))
```

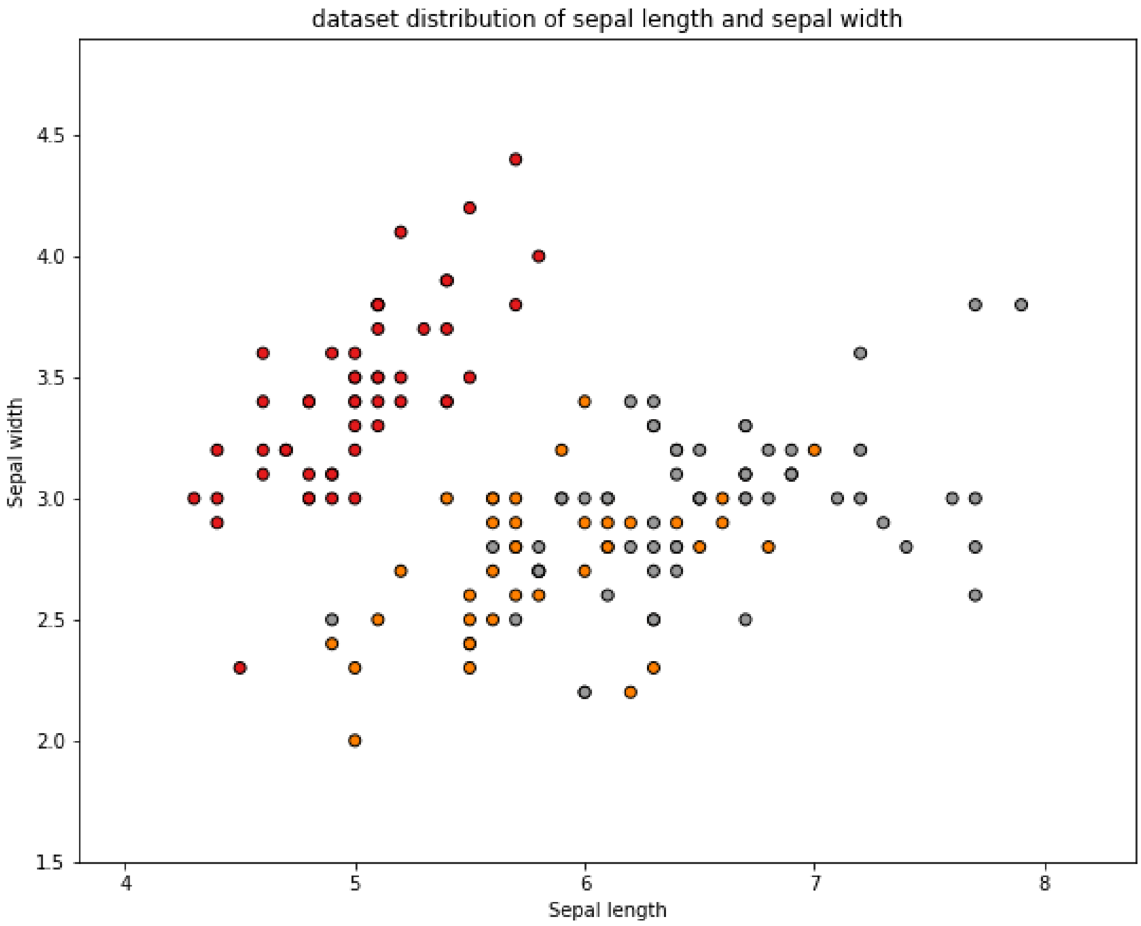
Visualize the iris dataset

In [14]:

```
from deeplearning import show_data_in_2d
```

In [15]:

```
show_data_in_2d(X, y)
```



Binary classification

In [16]:

```
# We will take the first 100 data entries as our dataset
# for the first 100 data entries only has class 0 and 1
X, y = X[:100], y[:100]
```

In [17]:

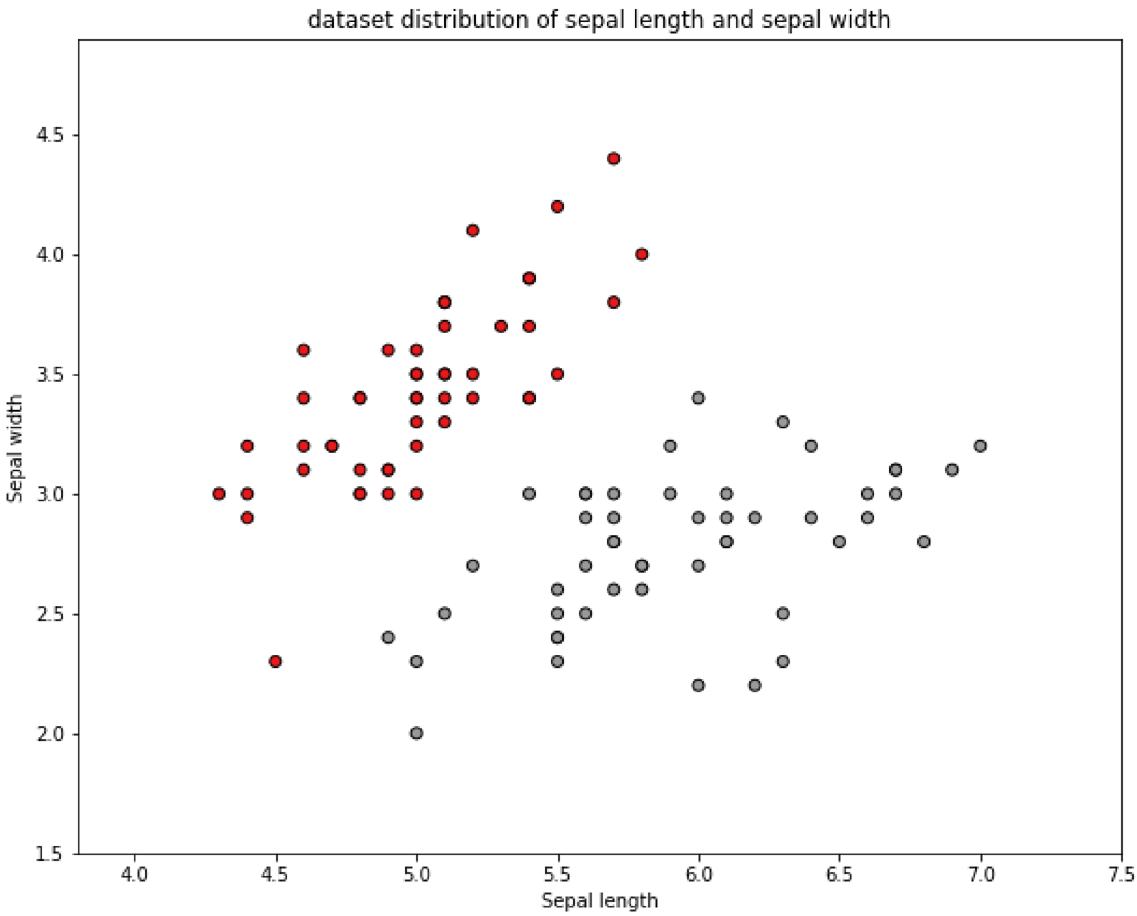
```
X.shape, y.shape
```

Out[17]:

((100, 4), (100,))

In [18]:

```
show_data_in_2d(X, y)
```



Create train_test_split for binary classification

In [19]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

In [20]:

```
X_train.shape, y_train.shape
```

Out[20]:

```
((70, 4), (70,))
```

In [21]:

```
X_test.shape, y_test.shape
```

Out[21]:

```
((30, 4), (30,))
```

Feature Scaling

In [22]:

```
# compute the mean and std for feature
f_mean, f_std = np.mean(X_train, axis=0), np.std(X_train, axis=0)
```

In [23]:

```
f_mean, f_std
```

Out[23]:

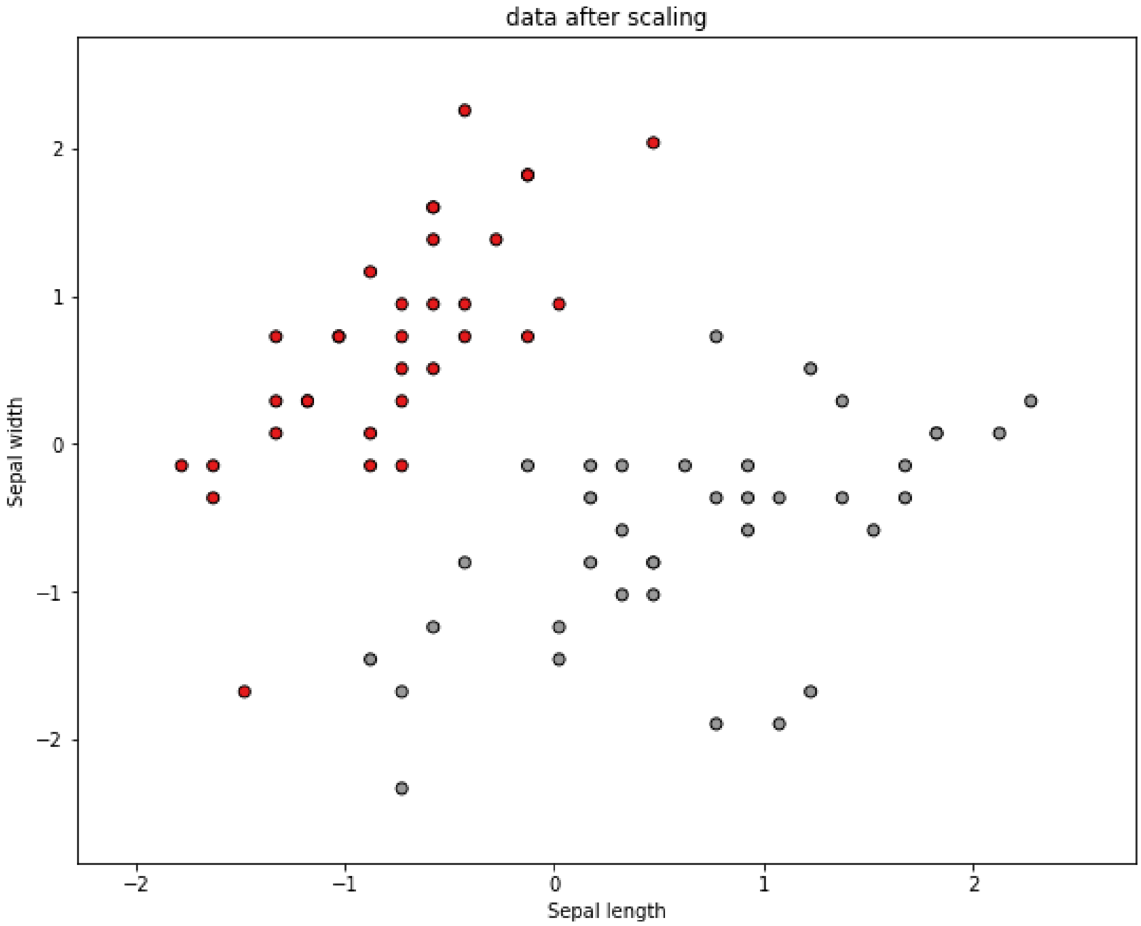
```
(array([5.48428571, 3.06714286, 2.9       , 0.80714286]),
 array([0.66561158, 0.4575467 , 1.40356688, 0.55531752]))
```

In [24]:

```
X_train = (X_train - f_mean) / f_std
X_test = (X_test - f_mean) / f_std
```

In [25]:

```
show_data_in_2d(X_train, y_train, title='data after scaling')
```



Initialze the classifier weight

In [26]:

```
# weight initialization using zero
theta = np.zeros((X_train.shape[1] + 1))
# weight initialization using random
# np.random.seed(42)
# theta = np.random.rand(X_train.shape[1] + 1, 1)
```

In [27]:

```
# the extra dimension is for the bias
theta
```

Out[27]:

```
array([0., 0., 0., 0., 0.])
```

In [28]:

```
# Concatenate X with a new dimension for bias
X_train = np.concatenate((np.ones((X_train.shape[0], 1)), X_train), axis=1)
X_test = np.concatenate((np.ones((X_test.shape[0], 1)), X_test), axis=1)
```

Forward pass, compute classifier output and cross entropy loss

compute $y_{\theta}(x)$ $$ y_{\theta}(x)=\frac{1}{1+e^{-\theta^T x}} $$

compute $J(\theta)$
$$ J(\theta)=\frac{1}{m}\sum_{i=1}^{m}Cost(y_{\theta}(x^{(i)}),t^{(i)}) $$

compute $Cost(y_{\theta}, t)$ (cross entropy)
$$ Cost(y_{\theta}, t)=-t log((y_{\theta}(x))-(1-t)log(1-(y_{\theta}(x))) $$

In [29]:

```
# compute h_{\theta}(x)
logits = np.dot(X_train, theta)
logits.shape
```

Out[29]:

```
(70,)
```

In [30]:

```
h = 1 / (1 + np.exp(-logits))
```

In [31]:

```
cross_entropy_loss = (-y_train * np.log(h) - (1 - y_train) * np.log(1 - h)).mean()
```

In [32]:

```
cross_entropy_loss
```

Out[32]:

```
0.6931471805599454
```

Backward pass, compute gradients and update the classifier's weight

compute the gradient $$ \frac{\partial J(\theta)}{\partial \theta}=\sum_{i=1}^{m}(y_{\theta}(x)-t^{(i)})x^{(i)} $$

update the weights $$ \theta_{j}^{new}=\theta_{j}^{old}-\alpha\frac{\partial J(\theta)}{\partial \theta_j} $$

In [33]:

```
gradient = np.dot((h - y_train), X_train) / y.size
```

In [34]:

```
gradient
```

Out[34]:

```
array([-0.02      , -0.2551252 ,  0.22914133, -0.33842349, -0.33558664])
```

In [35]:

```
# alpha = 0.01
theta = theta - 0.01 * gradient
# python provides a more consice code
# theta -= 0.01 * gradient
```

In [36]:

```
theta
```

Out[36]:

```
array([ 0.0002    ,  0.00255125, -0.00229141,  0.00338423,  0.00335587])
```

In [37]:

```
np.random.seed(21)
theta = np.random.rand(*theta.shape)
```

Put everything together and form a train loop

In [38]:

```
from deeplearning import plot_decision_regions
num_epoch = 1000
for epoch in range(num_epoch):
    # forward pass
    logits = np.dot(X_train, theta)
    h = 1 / (1 + np.exp(-logits))
    cross_entropy_loss = (-y_train * np.log(h) - (1 - y_train) * np.log(1 - h)).mean()

    # backward pass
    gradient = np.dot((h - y_train), X_train) / y.size
    theta = theta - 0.01 * gradient

    if epoch % 50 == 0:
        print('Epoch', epoch, 'loss:', cross_entropy_loss)
```

```
Epoch 0 loss: 0.7748346144585231
Epoch 50 loss: 0.5720708771728393
Epoch 100 loss: 0.440792665066672
Epoch 150 loss: 0.352983466314648
Epoch 200 loss: 0.29183884004925575
Epoch 250 loss: 0.2475511555793824
Epoch 300 loss: 0.2143272204118517
Epoch 350 loss: 0.18864210414163093
Epoch 400 loss: 0.16827383164332294
Epoch 450 loss: 0.15177139353644767
Epoch 500 loss: 0.13815554476377095
Epoch 550 loss: 0.12674549413545205
Epoch 600 loss: 0.11705525275702199
Epoch 650 loss: 0.10872966265972545
Epoch 700 loss: 0.10150372856409208
Epoch 750 loss: 0.09517604848333451
Epoch 800 loss: 0.08959101743916129
Epoch 850 loss: 0.0846266344888491
Epoch 900 loss: 0.08018597539572045
Epoch 950 loss: 0.07619111661215895
```

Inference the model on test set

In [39]:

```
h_test = 1 / (1 + np.exp(-np.dot(X_test, theta)))
```

In [40]:

```
h_test
```

Out[40]:

```
array([0.98681517, 0.93420641, 0.98212737, 0.06393632, 0.08129591,
       0.05677605, 0.01994854, 0.88132236, 0.06210332, 0.04913513,
       0.10888063, 0.05551006, 0.96132714, 0.04397681, 0.92297714,
       0.04082092, 0.98900512, 0.99332768, 0.04278287, 0.11432648,
       0.94430578, 0.90632   , 0.08109439, 0.02708437, 0.89170272,
       0.07446794, 0.0504435 , 0.92656275, 0.04721615, 0.98858739])
```

In [41]:

```
y_test
```

Out[41]:

```
array([1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1,
       0, 0, 1, 0, 0, 1, 0, 1])
```

In [42]:

```
((h_test > 0.5) == y_test).sum() / y_test.size
```

Out[42]:

```
1.0
```

Visualize the decision boundary

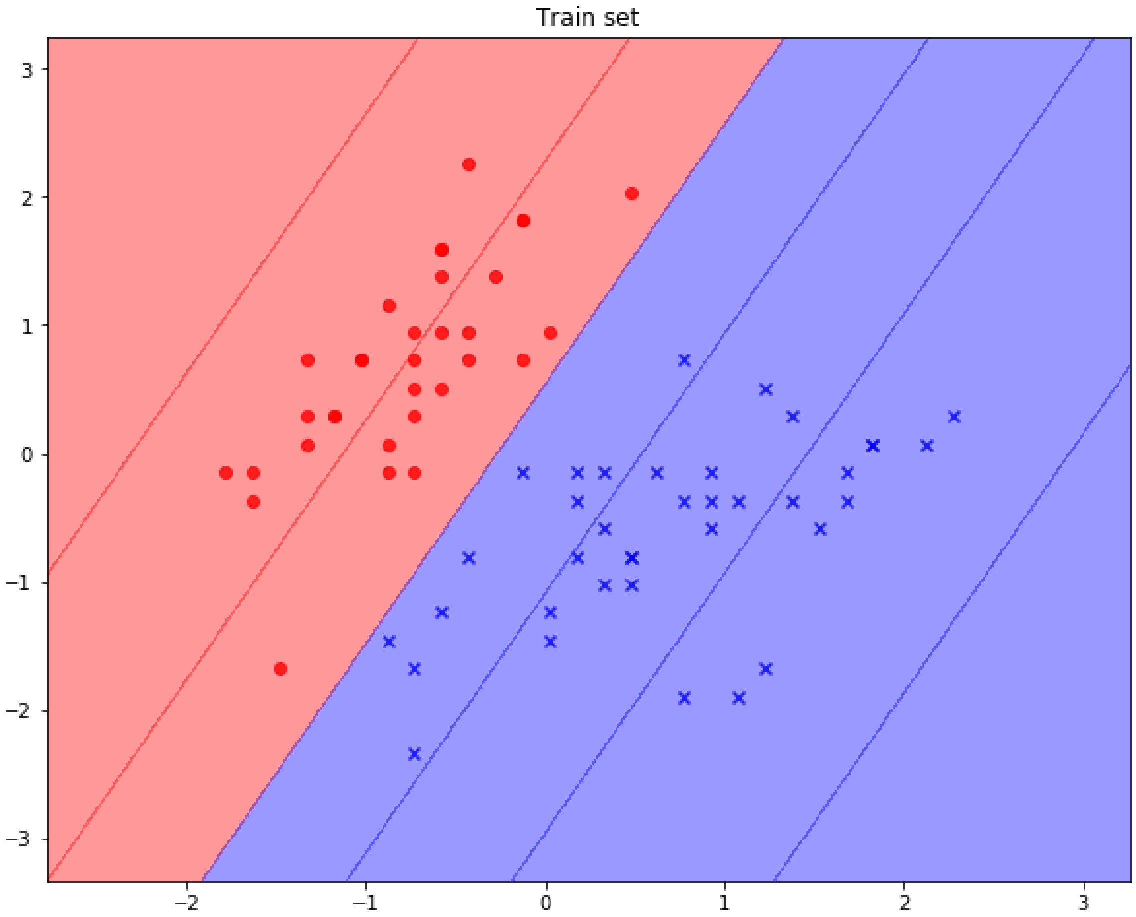
In [43]:

```
from deeplearning import plot_decision_regions
```

In [44]:

```
plot_decision_regions(X_train[:, 1:], y_train, theta[1:3])
plt.title('Train set')
plt.show()
```

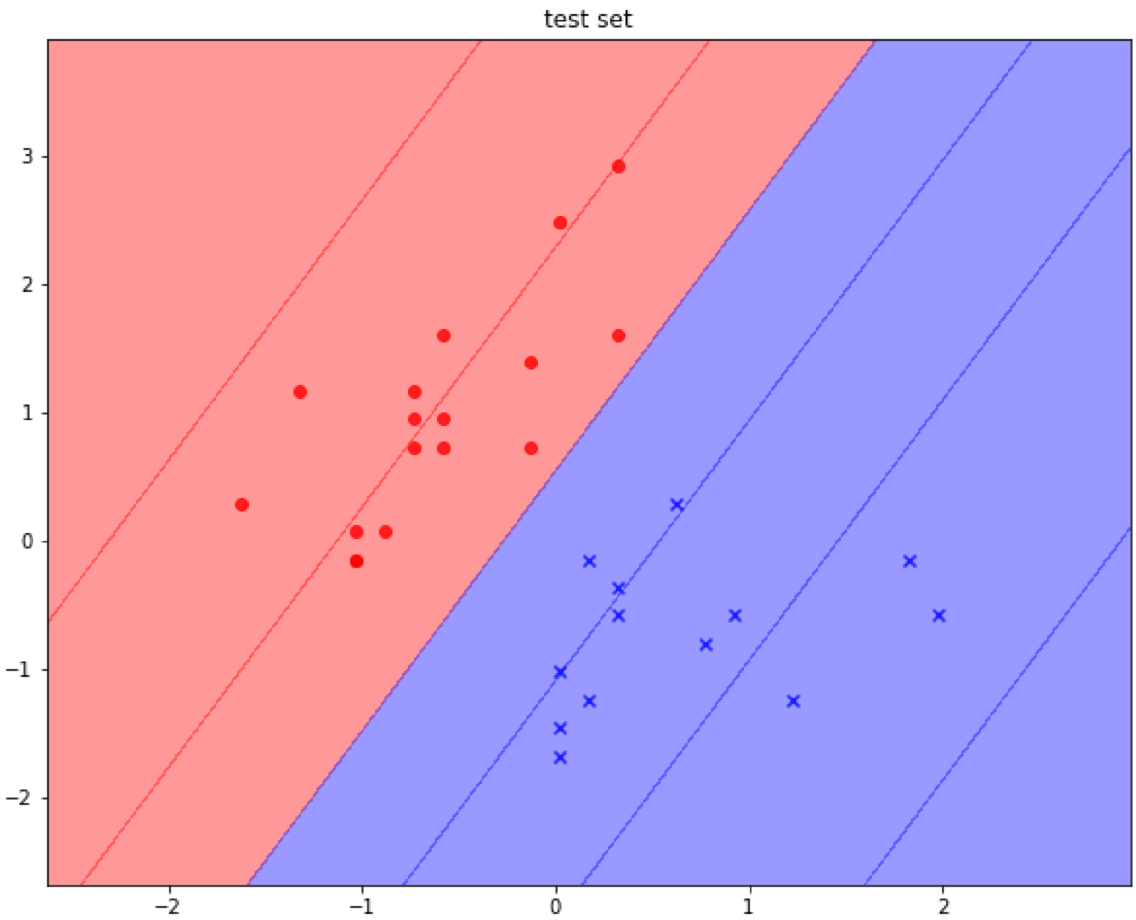
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with 'x' & 'y'. Please use a 2-D array with a single row if you really want to specify the same RGB or RGBA value for all points.



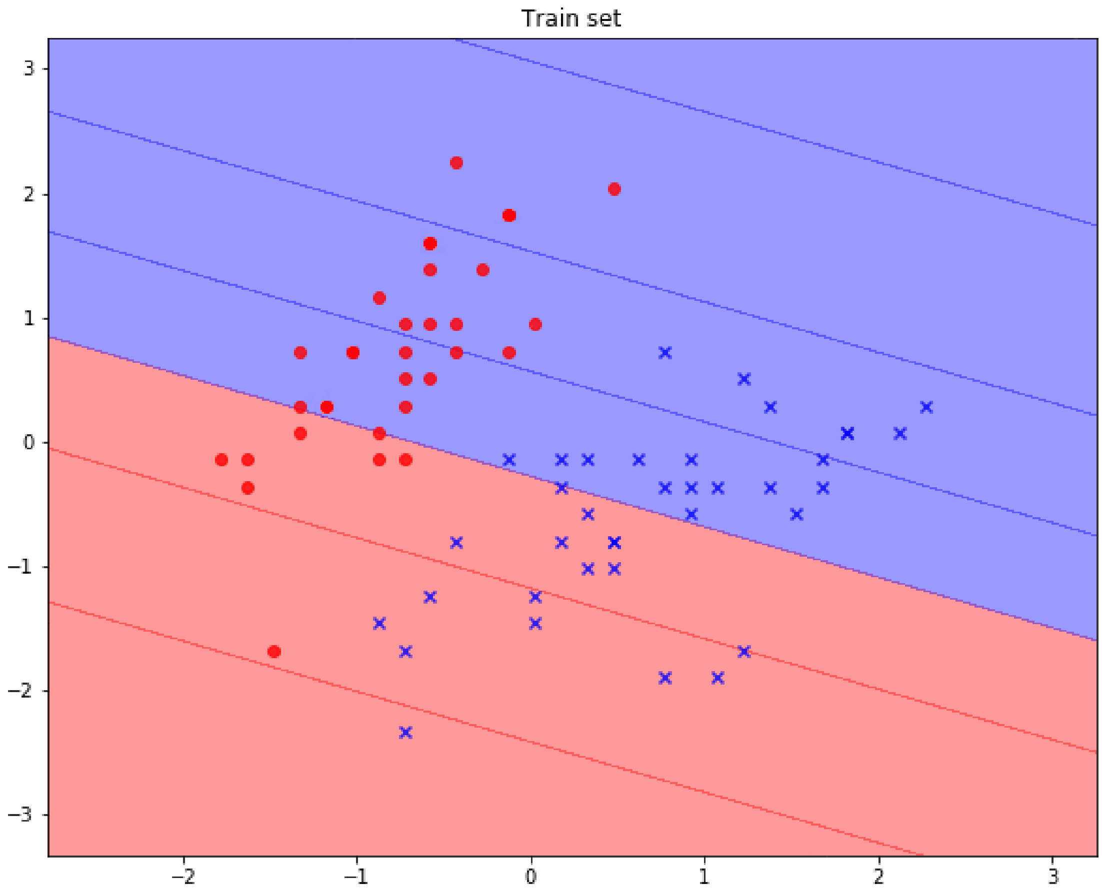
In [45]:

```
plot_decision_regions(X_test[:, 1:], y_test, theta[1:3])
plt.title('test set')
plt.show()
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with 'x' &
'y'.  Please use a 2-D array with a single row if you really want to specify the s
ame RGB or RGBA value for all points.
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avo
ided as value-mapping will have precedence in case its length matches with 'x' &
'y'.  Please use a 2-D array with a single row if you really want to specify the s
ame RGB or RGBA value for all points.
```



a GIF of how the model updates the weight and decision boundary



Using sklearn

In [46]:

```
from sklearn.linear_model import LogisticRegression
```

In [48]:

```
X_train.shape, y_train.shape
```

Out[48]:

```
((70, 5), (70,))
```

In [50]:

```
model = LogisticRegression()
model.fit(X_train, y_train)
```

```
C:\Users\ThinkPad\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:43
2: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a sol
ver to silence this warning.
  FutureWarning)
```

Out[50]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [51]:

```
h_test = model.predict_proba(X_test)
```

In [54]:

```
h_test[:10]
```

Out[54]:

```
array([[0.00244015, 0.99755985],
       [0.01121045, 0.98878955],
       [0.00698074, 0.99301926],
       [0.95842844, 0.04157156],
       [0.97970424, 0.02029576],
       [0.97963997, 0.02036003],
       [0.99555744, 0.00444256],
       [0.02841768, 0.97158232],
       [0.98554709, 0.01445291],
       [0.98530274, 0.01469726]])
```

In [55]:

```
h_test = h_test[:, 1]
```

In [56]:

```
h_test
```

Out[56]:

```
array([0.99755985, 0.98878955, 0.99301926, 0.04157156, 0.02029576,
       0.02036003, 0.00444256, 0.97158232, 0.01445291, 0.01469726,
       0.02532906, 0.03155805, 0.98860154, 0.00466937, 0.98082053,
       0.01045374, 0.99721175, 0.99812736, 0.02526217, 0.04673157,
       0.98305416, 0.95609298, 0.03318362, 0.01194769, 0.97116052,
       0.00683754, 0.0153187 , 0.97144192, 0.02489135, 0.99827021])
```

In [57]:

```
((h_test > 0.5) == y_test).sum() / y_test.size
```

Out[57]:

```
1.0
```

Multi-class classification

In [58]:

```
def get_classifier(X_train, y_train, num_epoch=1000, alpha=0.01):
    theta = np.zeros((X_train.shape[1]))
    for epoch in range(num_epoch):
        # forward pass
        logits = np.dot(X_train, theta)
        h = 1 / (1 + np.exp(-logits))
        cross_entropy_loss = (-y_train * np.log(h) - (1 - y_train) * np.log(1 - h)).mean()

        # backward pass
        gradient = np.dot((h - y_train), X_train) / y.size
        theta = theta - alpha * gradient
    return theta
```

In [59]:

```
def multi_classifier(X_train, y_train):
    num_class = np.unique(y_train)
    param = np.zeros((len(num_class), X_train.shape[1]))

    for i in num_class:
        label_t = np.zeros_like(y_train)
        num_class = np.unique(y_train)
        label_t[y_train == num_class[i]] = 1
        param[i, :] = get_classifier(X_train, label_t)

    return param
```

In [60]:

```
# get iris data
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

f_mean, f_std = np.mean(X_train, axis=0), np.std(X_train, axis=0)
X_train = (X_train - f_mean) / f_std
X_test = (X_test - f_mean) / f_std

X_train = np.concatenate((np.ones((X_train.shape[0], 1)), X_train), axis=1)
X_test = np.concatenate((np.ones((X_test.shape[0], 1)), X_test), axis=1)
```

In [61]:

```
params = multi_classifier(X_train, y_train)
```

In [62]:

```
def pred(param, X_test, y_test):
    f_size = X_test.shape
    l_size = y_test.shape
    assert (f_size[0] == l_size[0])

    logits = np.dot(X_test, np.transpose(param)).squeeze()
    prob = 1 / (1 + np.exp(-logits))

    pred = np.argmax(prob, axis=1)
    accuracy = np.sum(pred == y_test) / l_size[0] * 100

    return prob, pred, accuracy
```

In [63]:

```
_, preds, accu = pred(params, X_test, y_test)
print("Prediction: {}\n".format(preds))
print("Accuracy: {:.3f}%".format(accu))
```

```
Prediction: [1 0 2 2 2 0 1 2 1 1 2 0 0 0 0 2 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 2 0 0
2 2
 0 0 0 2 2 2 0 0]

Accuracy: 84.444%
```

In [ ]:

```

```