

Scikit-learn/sklearn

Ibrahim A. Hameed, PhD, Professor

ML IE500618

NTNU in Ålesund

Topics

- Sklearn
- Iris classification in sklearn (binary classification)
- Plot of decision boundary
- Split data into training and testing
- A perceptron for multiclass classification/one-vs.-rest (OVR) approach
- Logistic regression classifier
- Logistic regression for IRIS dataset
- Separable and inseparable dataset
- Confusion matrix
- Multilayer neural network (MLP)
- MLP classifier in sklearn
- MLP classifier in Keras/model.summary()

Scikit-learn

- Is a Python machine learning library offering a vast collection of machine learning algorithms from data processing, model selection, model training, fine tune, and evaluation.
- **sciPy Toolkit** developed by Google.
- <https://scikit-learn.org/stable/>

scikit-learn

Machine Learning in Python

[Getting Started](#)
[Release Highlights for 0.23](#)
[GitHub](#)

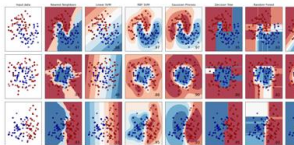
- Simple and efficient tools for predictive data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying which category an object belongs to.

Applications: Spam detection, image recognition.

Algorithms: SVM, nearest neighbors, random forest, and more...



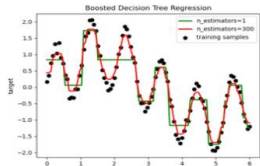
Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.

Algorithms: SVR, nearest neighbors, random forest, and more...




Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes

Algorithms: k-Means, spectral clustering, mean-shift, and more...



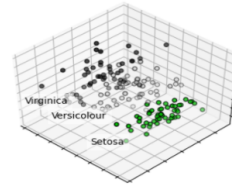
Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency

Algorithms: k-Means, feature selection, non-negative matrix factorization, and more...



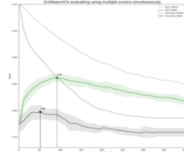
Examples

Model selection

Comparing, validating and choosing parameters and models.

Applications: Improved accuracy via parameter tuning

Algorithms: grid search, cross validation, metrics, and more...



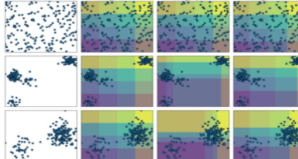
Examples

Preprocessing

Feature extraction and normalization.

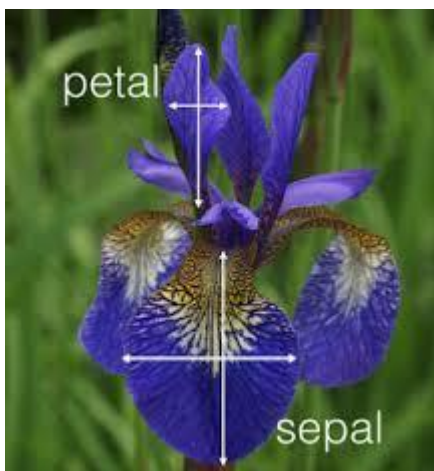
Applications: Transforming input data such as text for use with machine learning algorithms.

Algorithms: preprocessing, feature extraction, and more...



Examples

Iris dataset classification



	A	B	C	D	E
1	Sepal Length	Sepal Width	Petal Length	Petal Width	Class
2	5.1	3.5	1.4	0.2	Iris-setosa
3	4.9	3	1.4	0.2	Iris-setosa
4	4.7	3.2	1.3	0.2	Iris-setosa
5	4.6	3.1	1.5	0.2	Iris-setosa
6	5	3.6	1.4	0.2	Iris-setosa
7	5.4	3.9	1.7	0.4	Iris-setosa
8	4.6	3.4	1.4	0.3	Iris-setosa
9	5	3.4	1.5	0.2	Iris-setosa
10	4.4	2.9	1.4	0.2	Iris-setosa
11	4.9	3.1	1.5	0.1	Iris-setosa
12	5.4	3.7	1.5	0.2	Iris-setosa
13	4.8	3.4	1.6	0.2	Iris-setosa
14	4.8	3	1.4	0.1	Iris-setosa
15	4.3	3	1.1	0.1	Iris-setosa
16	5.8	4	1.2	0.2	Iris-setosa
17	5.7	4.4	1.5	0.4	Iris-setosa
18	5.4	3.9	1.3	0.4	Iris-setosa
19	5.1	3.5	1.4	0.3	Iris-setosa
20	5.7	3.8	1.7	0.3	Iris-setosa
21	5.1	3.8	1.5	0.3	Iris-setosa
22	5.4	3.4	1.7	0.2	Iris-setosa
23	5.1	3.7	1.5	0.4	Iris-setosa
24	4.6	3.6	1	0.2	Iris-setosa
25	5.1	3.3	1.7	0.5	Iris-setosa

Implementation

```
# implement a perceptron to classify iris using Scikit-Learn
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.linear_model import Perceptron
from sklearn.preprocessing import StandardScaler
```

```
iris = datasets.load_iris()
x = iris.data[0:100,[2,3]]    # petal length and petal width
y = iris.target[0:100]
```

```
print(x[0:5,:])
print('Class labels:', np.unique(y))
labels = ['setosa', 'versicolor', 'virginica']
```

```
# standarization
sc = StandardScaler()
sc.fit(x)    # to estimate mean and standard deviation
xstd = sc.transform(x)
print(xstd[0:5,:])
```

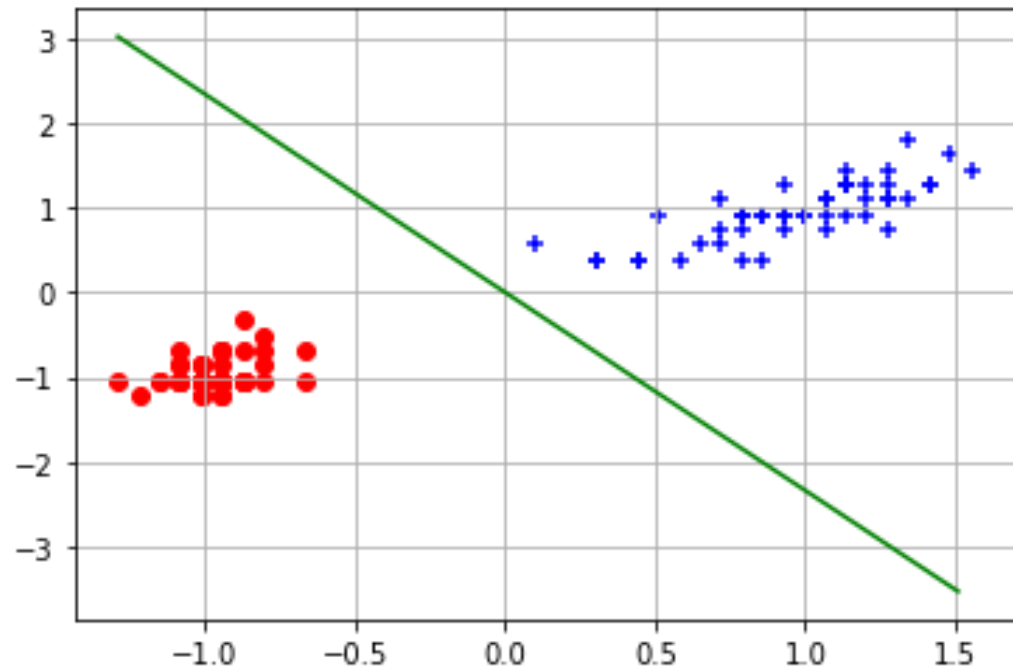
```
# import a model
#!pip install ....
model = Perceptron(max_iter=100, eta0=0.01, random_state=1)
model.fit(xstd, y)
y_pred=model.predict(x)

print(model.coef_, model.intercept_)

error=np.mean((y-y_pred)**2)
print(error)
```

```
#plot decision line
plt.scatter(xstd[y==0,0],xstd[y==0,1], color='red', marker='o',label=labels[0])
plt.scatter(xstd[y==1,0],xstd[y==1,1], color='blue', marker='+',label=labels[1])

xx = np.arange(xstd[:,0].min(),xstd[:,0].max(),0.1)
yy = -model.coef_[0,0]/model.coef_[0,1] * xx - model.intercept_/model.coef_[0,1]
plt.plot(xx,yy,'g-')
plt.grid()
plt.show()
```



Plot decision region

```
# Another way for plotting the decision line/region
def plot_decision_region(x, y):
    x1 = np.arange(x[:,0].min()-1, x[:,0].max()+1,0.1)
    x2 = np.arange(x[:,1].min()-1,x[:,1].max()+1, 0.1)

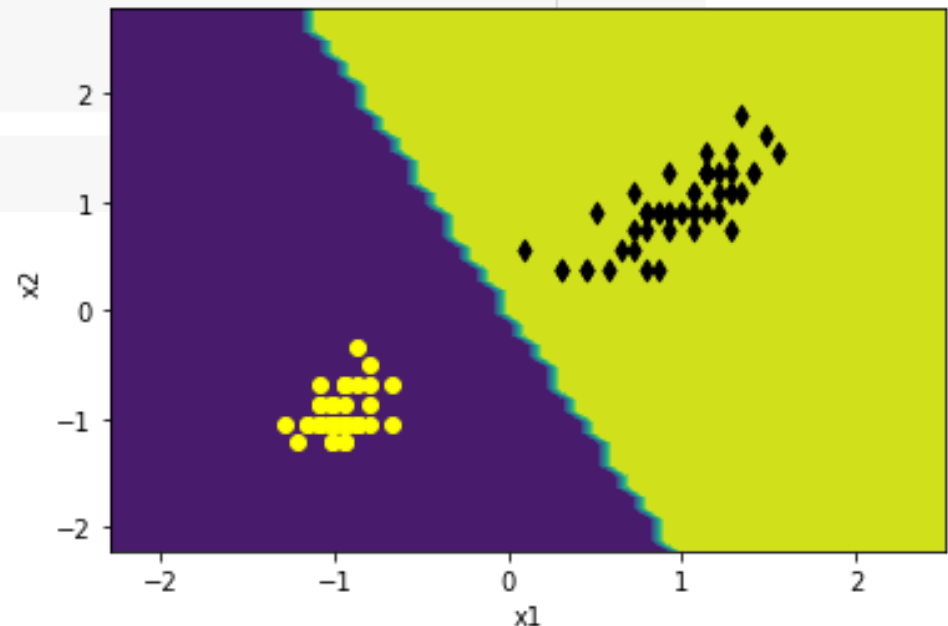
    xg1, xg2 = np.meshgrid(x1, x2)

    #print(xg2.flatten().T) # ravel()/flatten() returns a flattened one-dimensional array
    z = model.predict(np.array([xg1.ravel(), xg2.ravel()]).T)

    plt.contourf(xg1, xg2, z.reshape(xg1.shape))

    #plot the scatter plot of the dataset
    plt.scatter(x[y==0,0],x[y==0,1],color='yellow', marker='o' )
    plt.scatter(x[y==1,0],x[y==1,1],color='black', marker='d' )
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.show()
```

```
plot_decision_region(x,y)
```



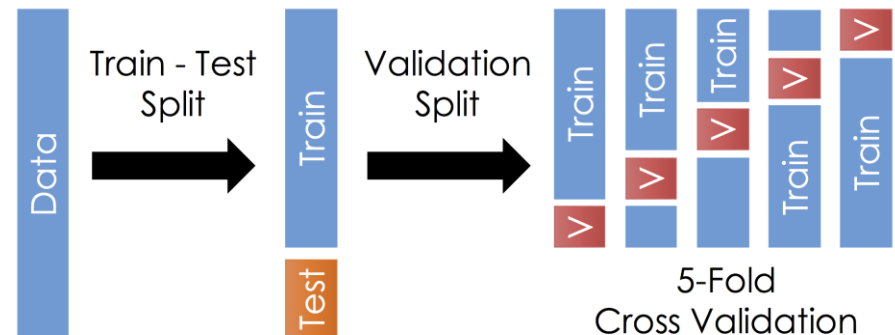
Splitting the dataset into training and testing sets

- It is important to evaluate the trained model performance on unseen data

```
# split the dataset into training and testing sets using (train_test_split) function
# this function shuffles the training sets internally before splitting
from sklearn.model_selection import train_test_split

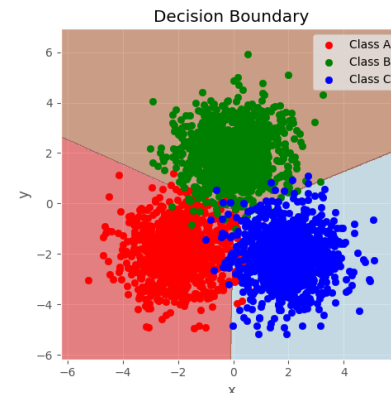
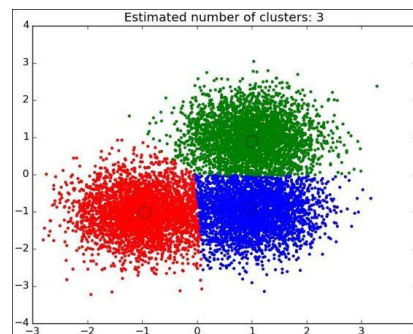
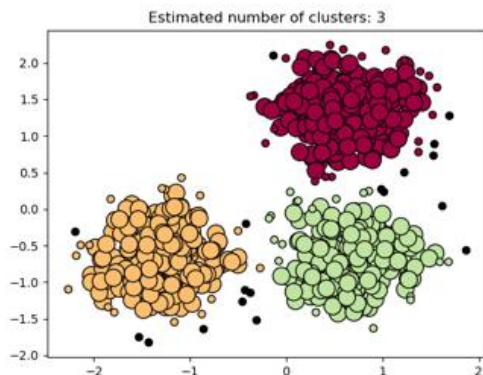
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=0.3, random_state=1)
print(ytest)
print('proportion of class labels in train and test sets:\n', np.bincount(ytrain), np.bincount(ytest))
#stratify: to have the same proportional of class labels in both training and testing sets (balanced sets)
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=0.3, random_state=1, stratify=y)
print('proportion of class labels in train and test sets:\n', np.bincount(ytrain), np.bincount(ytest))
```

```
[1 1 0 1 1 0 0 1 1 1 1 0 1 1 1 0 0 0 1 0 0 1 1 1 0 0 1 0 0 0]
proportion of class labels in train and test sets:
[36 34] [14 16]
proportion of class labels in train and test sets:
[35 35] [15 15]
```

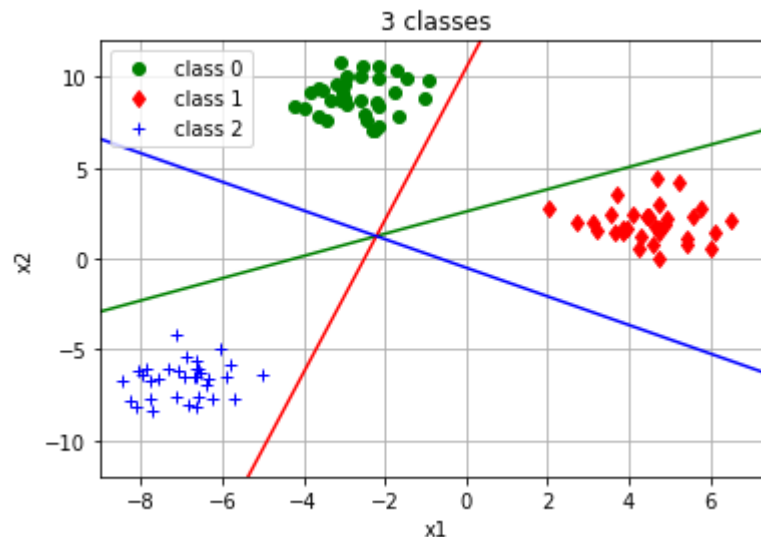
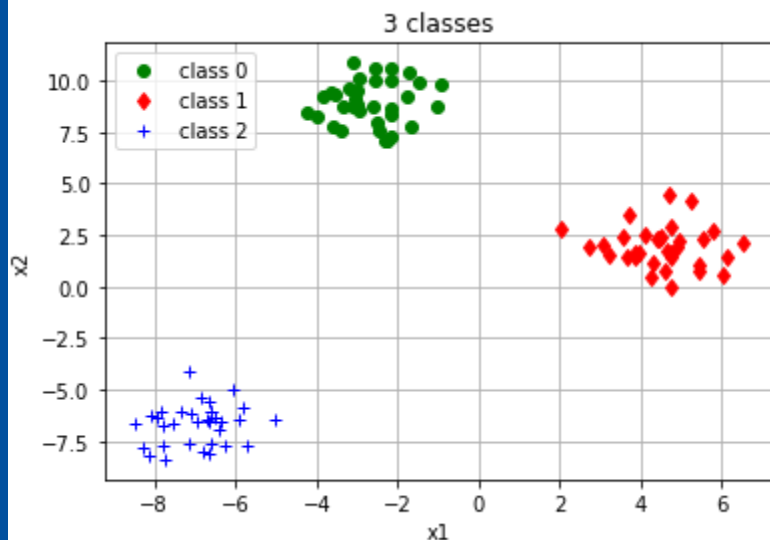


Linear models for multiclass classification

- Many linear classification models are for binary classification only, and do not extend naturally to the multiclass case (except for logistic regression)
- A common technique to extend a binary classification algorithm to a multiclass classification algorithm is the one-vs.-rest (OVR) approach.
- In OVR approach, a binary model is learned for each class that tries to separate that class from all of the other classes, resulting in as many binary models as there are classes.
- Having one binary classifier per class results in having one vector of coefficients (w) and one intercept (b) for each class.
- Sklearn provides a sample dataset generator to help you to create fast and easy to use a custom dataset.
- `make_blobs` – generates isotropic Gaussian blobs for clustering.



Make_blobs dataset: logistic regression



one-vs.-rest (OVR)
approach

	w1	w2	b
Class 0	-0.38	0.63	-1,63
Class 1	0.74	-0.18	1.86
Class 2	-0.36	-0.45	-0.23

```
#make_blobs dataset for multiclass classification
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
```

```
x, y = make_blobs(random_state=42)
```

```
print(y.shape)
print(np.bincount(y))
```

```
plt.plot(x[y==0,0], x[y==0,1], 'go', label='class 0')
plt.plot(x[y==1,0], x[y==1,1], 'rd', label='class 1')
plt.plot(x[y==2,0], x[y==2,1], 'b+', label='class 2')
```

```
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('3 classes')
plt.legend(loc='upper left')
plt.grid()
plt.show()
```

```
# build a model
model = LogisticRegression()
model.fit(x,y)
y_pred = model.predict(x)
```

```
print(model.coef_)
print(model.intercept_)
```

```
# plot decision lines
plt.plot(x[y==0,0], x[y==0,1], 'go', label='class 0')
plt.plot(x[y==1,0], x[y==1,1], 'rd', label='class 1')
plt.plot(x[y==2,0], x[y==2,1], 'b+', label='class 2')
```

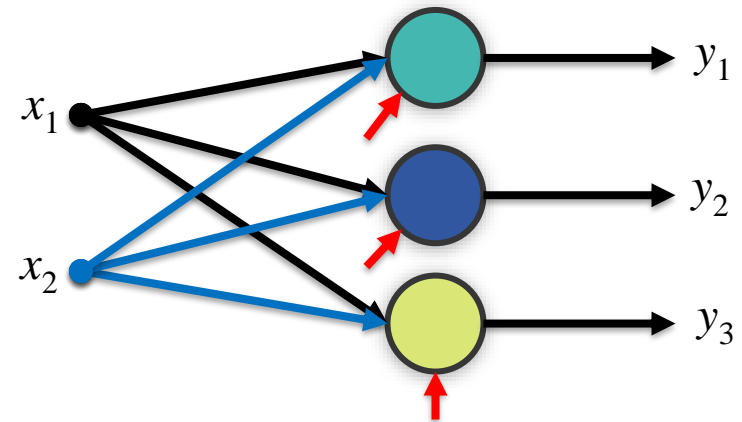
```
x1 = np.linspace(-10, 10)
color = ['g-', 'r-', 'b-']
```

```
for coef, intercept, color in zip(model.coef_, model.intercept_, color):
    x2 = -(coef[0]*x1+intercept)/coef[1]
    plt.plot(x1, x2, color)
```

```
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('3 classes')
plt.legend(loc='upper left')
plt.ylim([-12.0, 12.0])
plt.xlim([-9.0, 7.5])
plt.grid()
plt.show()
```

Extending a perceptron for multiclass classification

- **Iris dataset:**
 - 2 input feature: [sepal length (cm), sepal width (cm), **petal length** (cm), and **petal width** (cm)]
 - 3 classes (**setosa**, **versicolor**, and **virginica**)



Weights		Bias
-0.00220757	-0.00237113	-0.002
0.00540865	-0.00210787	-0.002
0.00628849	0.00683785	-0.008

Implementation

```
# In this example we will use a perceptron from sklearn to classify three iris classes (setosa, versicolor, and virginica)
# upload python libraries
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
```

```
# upload your iris dataset
iris = datasets.load_iris()
# select two features
x = iris.data[:, [2, 3]] # petal length and petal width
y = iris.target
print(np.unique(y))
print(iris.target_names)
print(iris.feature_names)
```

```
[0 1 2]
['setosa' 'versicolor' 'virginica']
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

```
# standardize your feature vector
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
sc.fit(x)
xstd=sc.transform(x)
print(x[1:2,:], xstd[1:2,:])
```

```
[[1.4 0.2]] [[-1.34022653 -1.3154443 ]]
```

```
#split dataset into training and testing sets
from sklearn.model_selection import train_test_split
x = xstd
xtrain, xtest, ytrain, ytest = train_test_split(x,y,test_size=0.3, random_state=1,stratify=y)

print(np.bincount(ytrain), np.bincount(ytest))
```

```
[35 35 35] [15 15 15]
```

```
# select a model for training
from sklearn.linear_model import Perceptron
model = Perceptron(max_iter=20, eta0=0.002, random_state=1)
model.fit(xtrain, ytrain)
ypred = model.predict(xtest)
missclassified = (ytest != ypred).sum()
print('missclassified samples = ', missclassified)
print('Model accuracy = ', (len(ytest)-missclassified)/len(ytest)*100)
print('Model accuracy = ', model.score(xtest, ytest)*100, '%')

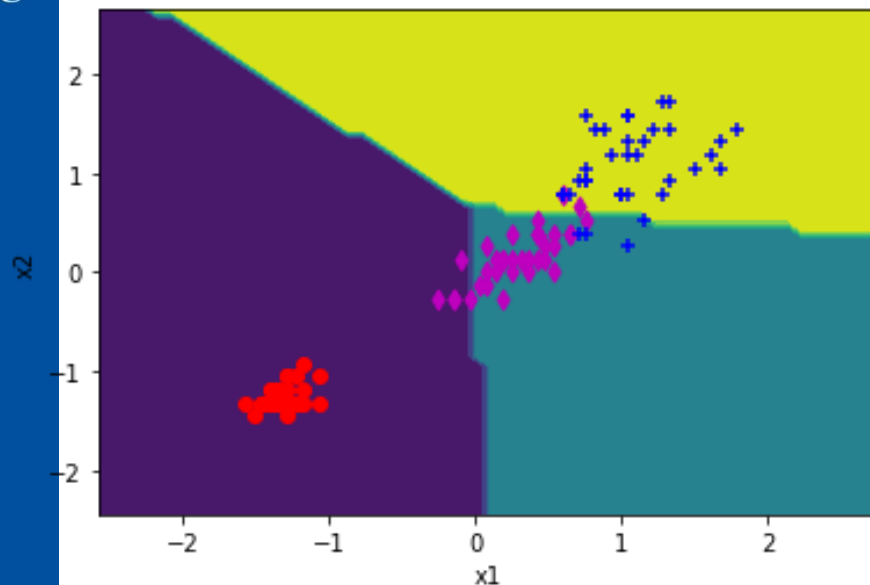
print('weights: \n', model.coef_)
print('bias : \n', model.intercept_)
```

```
missclassified samples = 2
Model accuracy = 95.55555555555556
Model accuracy = 95.55555555555556 %
weights:
[[-0.00220757 -0.00237113]
 [ 0.00540865 -0.00210787]
 [ 0.00628849  0.00683785]]
bias :
[-0.002 -0.002 -0.008]
```

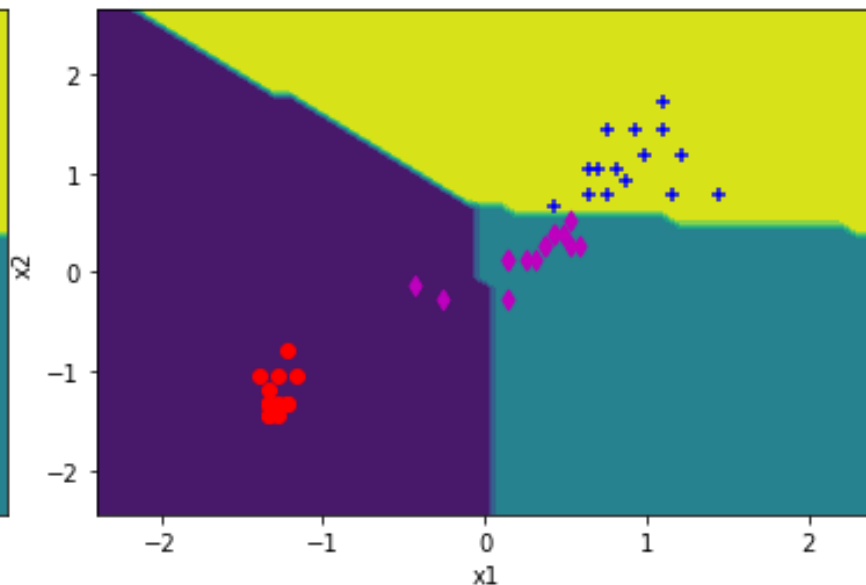
```
# plot decision regions in 2D space
x1 = np.arange(x[:,0].min()-1, x[:,0].max()+1, 0.1)
x2 = np.arange(x[:,1].min()-1, x[:,1].max()+1, 0.1)
x1g, x2g = np.meshgrid(x1, x2)
z = model.predict(np.array([x1g.flatten(), x2g.flatten()]).T)
plt.contourf(x1g, x2g, z.reshape(x1g.shape))
plt.scatter(x[y==0,0],x[y==0,1],color='red', marker='o')
plt.scatter(x[y==1,0],x[y==1,1],color='m', marker='d')
plt.scatter(x[y==2,0],x[y==2,1],color='blue', marker='+')
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```

Decision regions

TRAIN



TEST

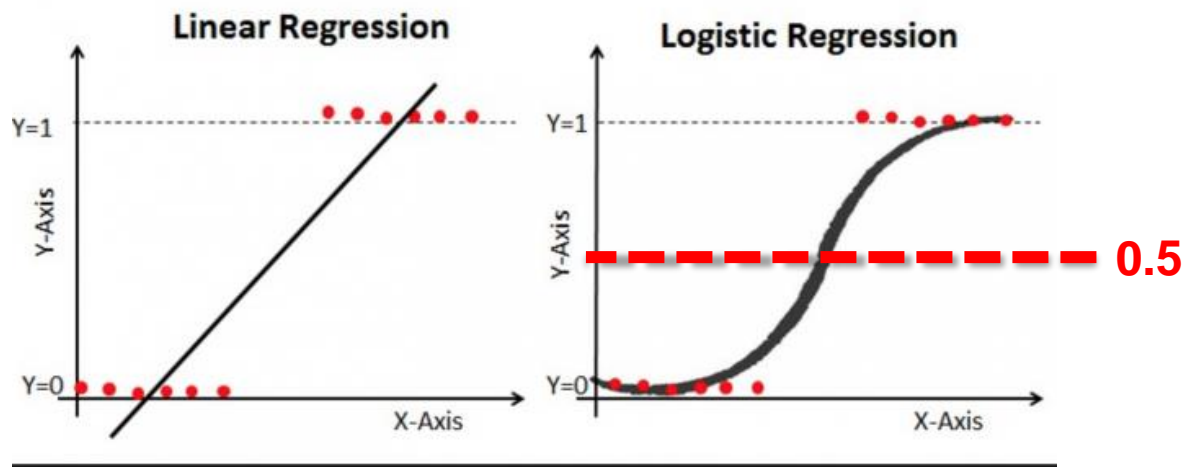


	Misclassified samples	Model accuracy
Train	11	89.52%
Test	2	95.55%

- Notes that a perceptron will never converge if the classes are not perfectly linearly separable.
- In iris dataset, there will be always at least one misclassified sample present in each epoch.

Modeling class probabilities via logistic regression

- Logistic regression is one of the most popular machine learning algorithms in industry.
- It is a classification model that performs very well for linearly separable binary classes (YES/NO, TRUE/FALSE) instead of predicting something continuous.
- Instead of fitting a line to the data, logistic regression fits an 'S' shaped logistic function
- Although it gives the probability as a number between 0 and 1, it is usually used for classification using a threshold value.
- Examples: ham/spam email, fraudulent (yes/no) online transactions, etc.



Logistic regression

$$y \in \begin{cases} 1 & \text{Positive Class (Yes / True)} \\ 0 & \text{Negative Class (No / False)} \end{cases}$$

- It can also be extended into multiclass classification

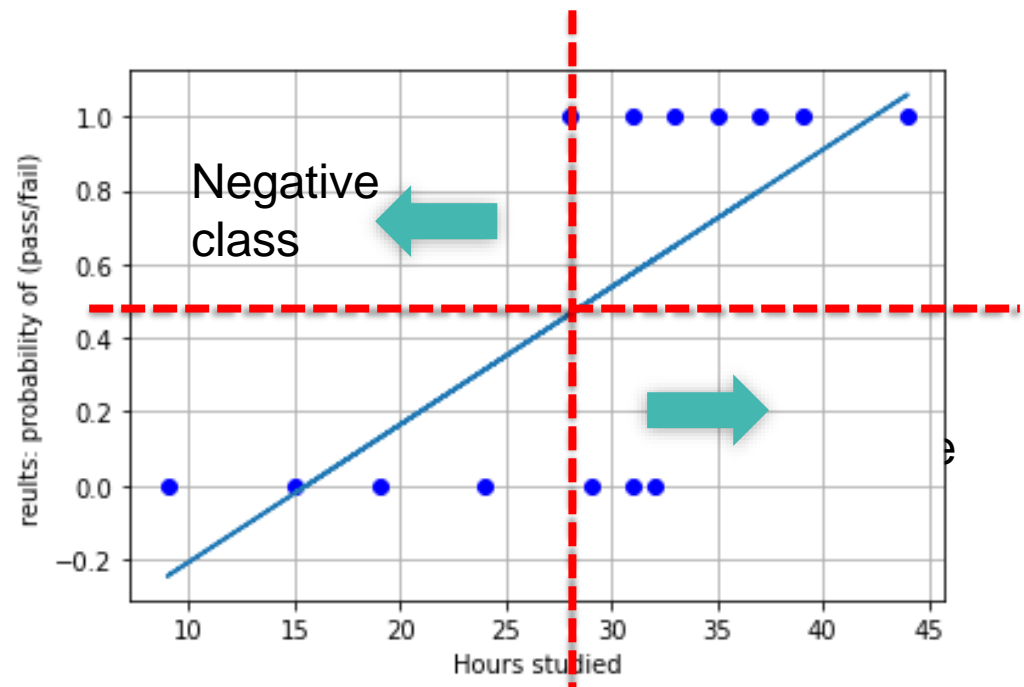
$$y \in \{0, 1, 2, 3\}$$

It is all about likelihood of an event: hours studied to pass example

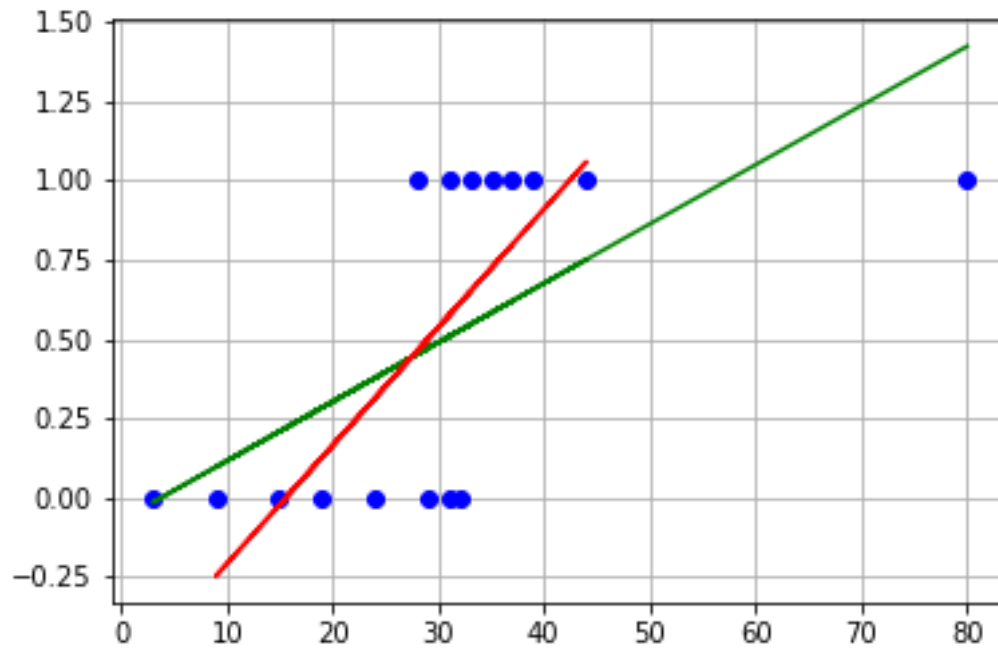
Hours studied	Exam result (1=pass, 0=fail)
29	0
15	0
33	1
28	1
39	1
44	1
31	1
19	0
9	0
24	0
32	0
31	0
37	1
35	1

- Can we fit a straight line?

```
m, b = np.polyfit(x, y, 1)
plt.plot(x, m*x+b)
plt.show()
```



Adding outliers



Fitting a logistic regression model: fit a sigmoid/logistic function

$$net = w^T x = wx + b = 0.4x - 12.05 \quad g(net) = \frac{1}{1 + e^{-net}} = \frac{e^{net}}{1 + e^{net}}$$

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

```
# fit a logistic regression model
```

```
logreg = LogisticRegression()
```

```
# fit the model with data
```

```
logreg.fit(x.reshape(-1,1),y.reshape(-1,1))
```

```
print(logreg.coef_)
```

```
print(logreg.intercept_)
```

```
xx = np.arange(5, 50,0.1).reshape(-1,1)
```

```
y_pred = logreg.predict(xx)
```

```
def sig(x):
```

```
    return(np.exp(x)/(1+np.exp(x)))
```

```
net = logreg.intercept_ + logreg.coef_ * xx
```

```
plt.plot(x,y,'bd')
```

```
plt.plot(xx, sig(net), 'g-')
```

```
plt.plot(xx,y_pred,'r-',LineWidth=3.0)
```

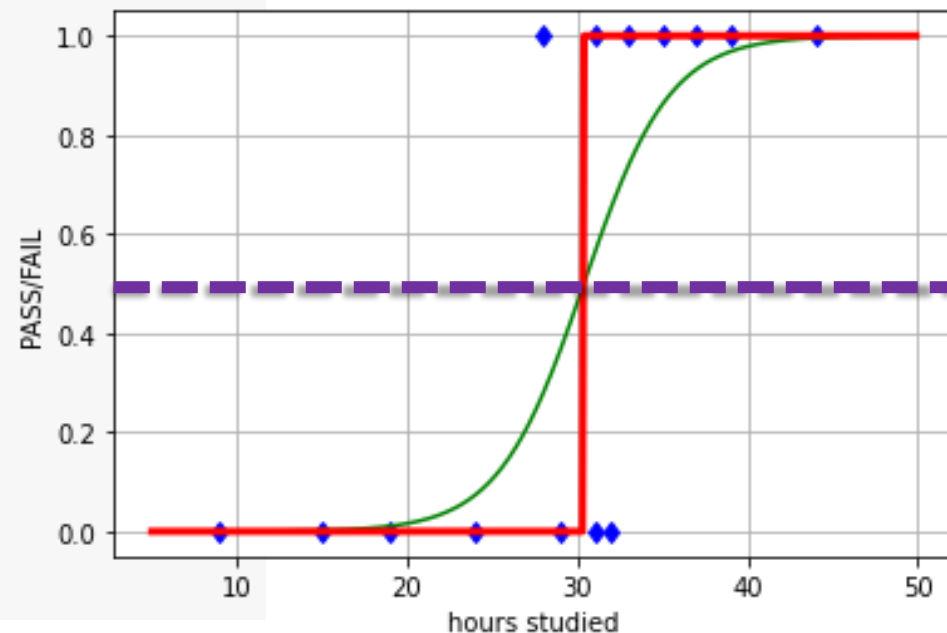
```
plt.grid()
```

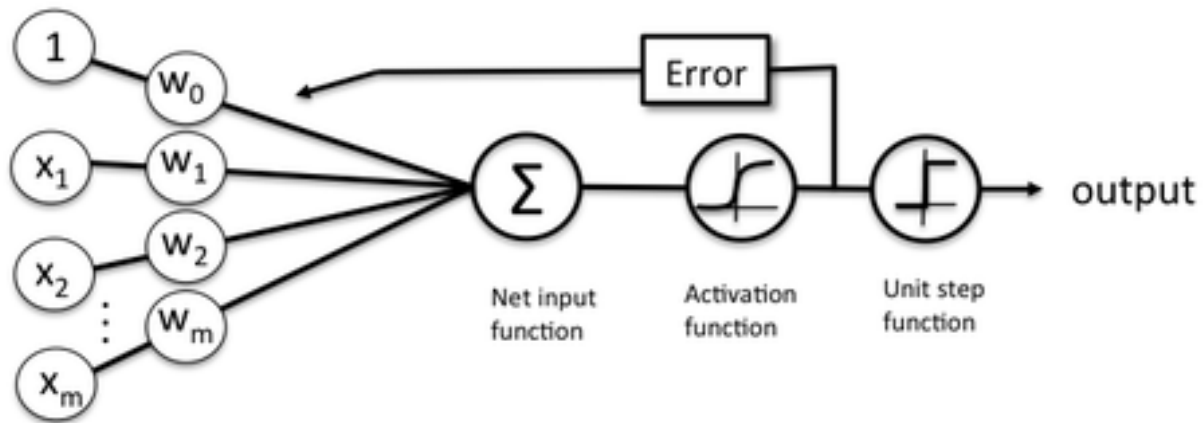
```
plt.show()
```

Hypothesis:

$y=1$ when $g(w^T x) \geq 0.5$

$y=0$ when $g(w^T x) < 0.5$

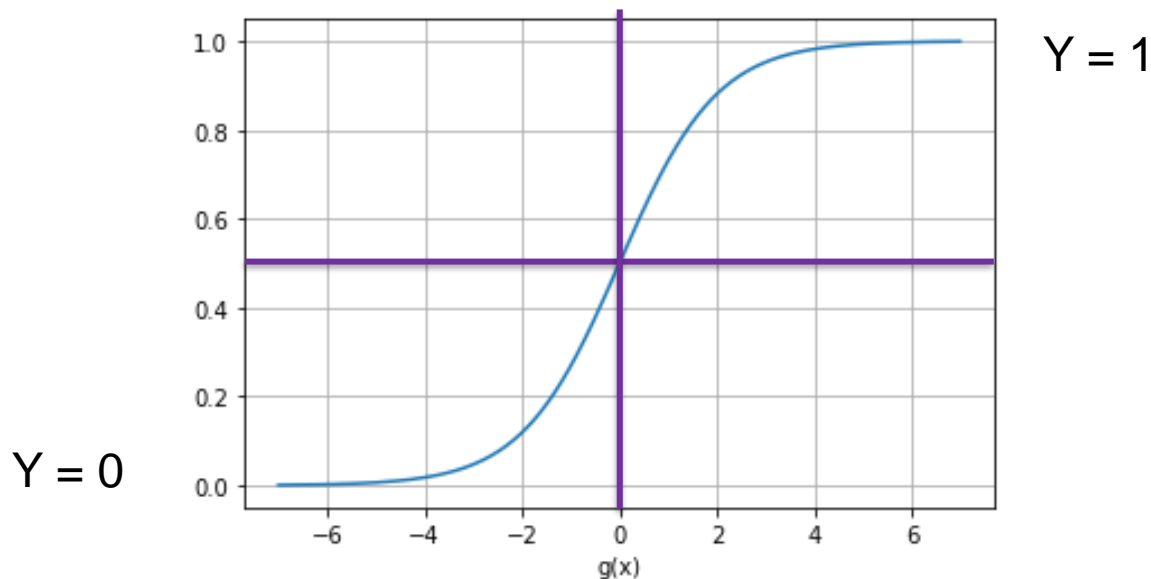




Schematic of a logistic regression classifier.

https://github.com/ibribr/ML/blob/master/logistic_regression.ipynb

Decision boundary



$$p(y=1) + p(y=0) = 1$$

$$p(y=1) = 1 - p(y=0)$$

$$y = g(x) \rightarrow 1 \text{ as } x \rightarrow \infty (1 / (1 + 0) = 1)$$

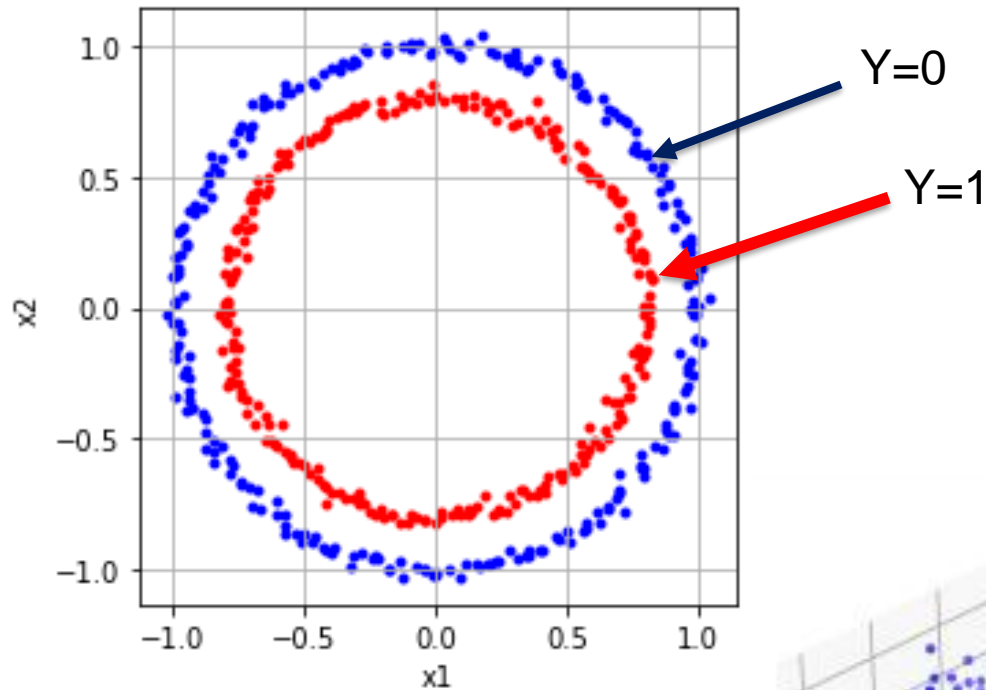
$$y = g(x) \rightarrow 0 \text{ as } x \rightarrow -\infty (1 / (1 + \infty) = 0)$$

Hypothesis:

$$y = 1 \text{ when } g(w^T x) \geq 0.5 \rightarrow w^T x \geq 0$$

$$y = 0 \text{ when } g(w^T x) < 0.5 \rightarrow w^T x < 0$$

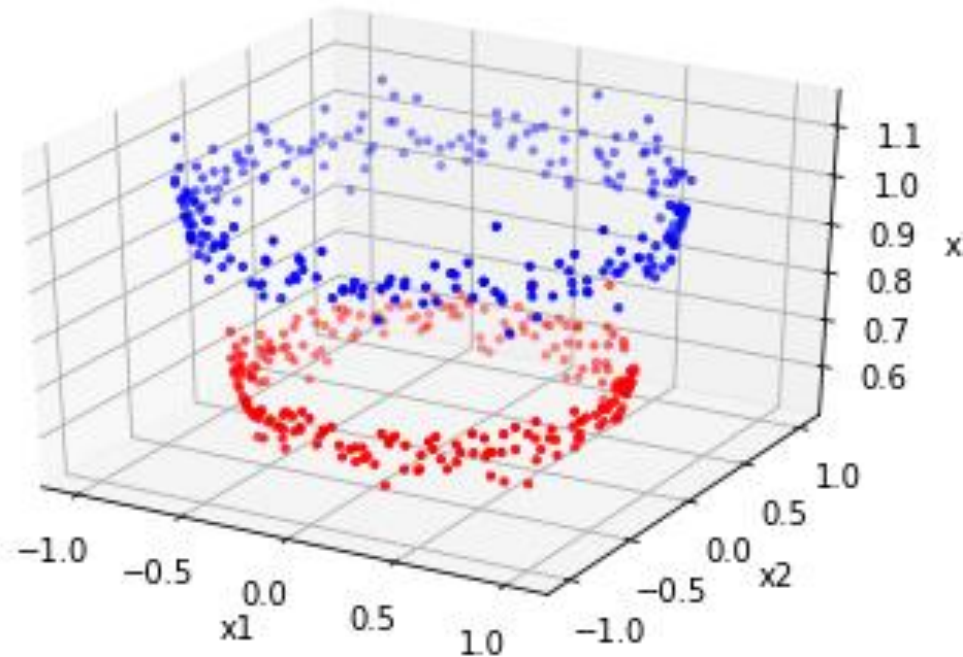
Linearly inseparable dataset



Adding a new dimension
(feature) of higher order

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$x_3 = x_1^2 + x_2^2$$

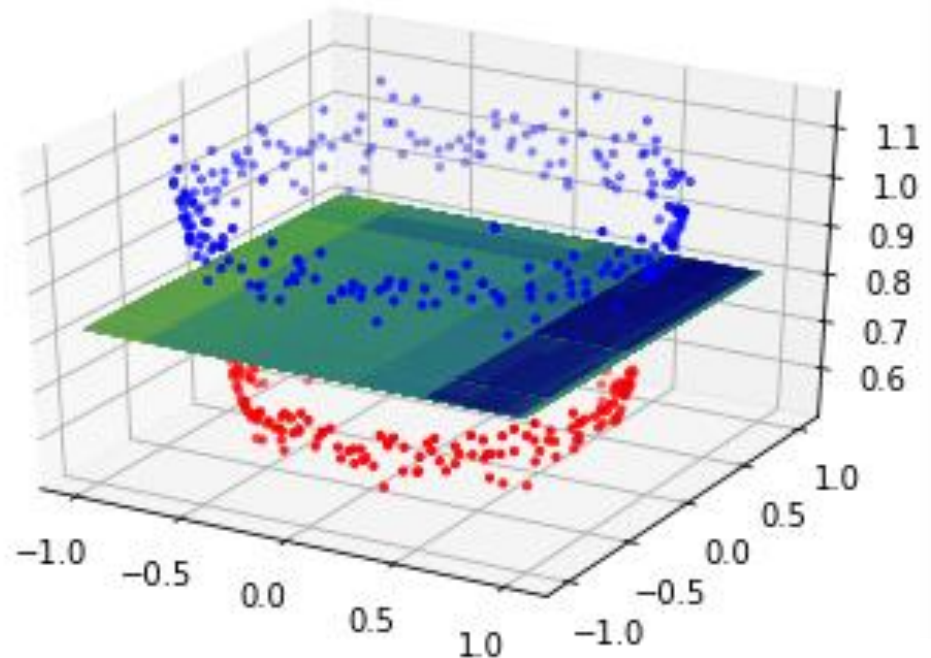


Fit a logistic regression model

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$x_3 = x_1^2 + x_2^2$$

$$net = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$



Logistic regression: Iris example

```
# implement a perceptron to classify iris using Scikit-Learn
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
```

```
iris = datasets.load_iris()
x = iris.data
y = iris.target

print(x[0:5,:])
print('Class labels:', np.unique(y))
labels = ['setosa', 'versicolor', 'virginica']

# standarization
sc = StandardScaler()
sc.fit(x) # to estimate mean and standard deviation
x = sc.transform(x)
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3. 1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5. 3.6 1.4 0.2]]
Class labels: [0 1 2]
```

```
# import a model
#!pip install ....
model = LogisticRegression(C=1, random_state=1)
model.fit(x, y)
y_pred=model.predict(x)
```

```
print(model.coef_, model.intercept_)
```

```
error=np.mean((y-y_pred)**2)
```

```
print(error)
```

```
print(np.sum(y!=y_pred))
```

```
[[ -1.07404149  1.16006342 -1.93062866 -1.81168873]
 [  0.58780051 -0.36182377 -0.36346274 -0.82619289]
 [  0.48624098 -0.79823965  2.2940914   2.63788161]] [-0.20531681  2.07486525 -1.86954844]
0.026666666666666667
4
```

Assignment

- Implement logistic regression
- Convert Adaline into a logistic regression
- Training set (m-examples)

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}, y \in \{0, 1\}$$

- Hypothesis:

$$\phi_w(x) = \frac{1}{1 + e^{-net}} = \frac{1}{1 + e^{-w^T x}} \quad (\text{notes : } net = w^T x + b = w^T x)$$

- Cost function:

$$J(w) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (\phi_w(x^{(i)}) - y^{(i)})^2 = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (\phi(net) - y^{(i)})^2$$

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(net) \geq 0.5 \\ 0 & \text{otherwise} \end{cases} = \begin{cases} 1 & \text{if } net = w^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$J(\phi(net), y) = \begin{cases} -\log(\phi(net)) & \text{if } y = 1 \\ -\log(1 - \phi(net)) & \text{if } y = 0 \end{cases}$$

$$J(\phi(net), y) = -y \log(\phi(net)) - (1 - y) \log(1 - \phi(net))$$

- Update the weights:

$$w_j = w_j + \Delta w_j$$

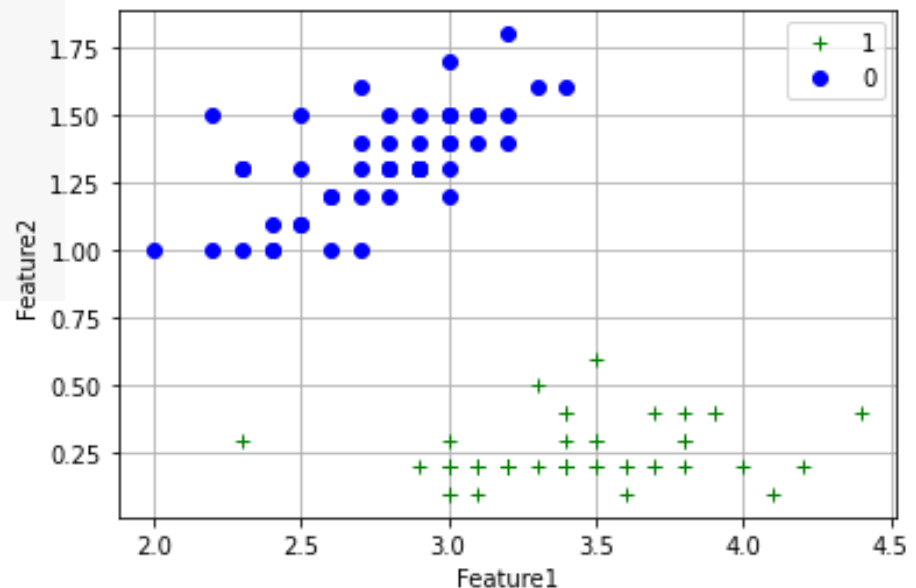
$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^m (y^{(i)} - \phi(w^T x_j^{(i)})) x_j^{(i)}$$

For Iris classification

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
```

```
iris = load_iris()
x = iris.data[1:100,[1,3]]
y = iris.target[1:100]
list(iris.target_names)
print(x.shape)
```

```
plt.plot(x[y==0,0],x[y==0,1], 'g+', label='1')
plt.plot(x[y==1,0],x[y==1,1], 'bo', label='0')
plt.legend(loc='upper right')
plt.xlabel('Feature1')
plt.ylabel('Feature2')
plt.grid()
plt.show()
```



A perceptron (/Adaline) with Gaussian activation function

```
# implement logistic regression classified
class logistic_regression(object):
    def __init__(self, epochs = 100, eta = 0.1, random_state=1):
        self.epochs = epochs
        self.eta = eta
        self.random_state = random_state

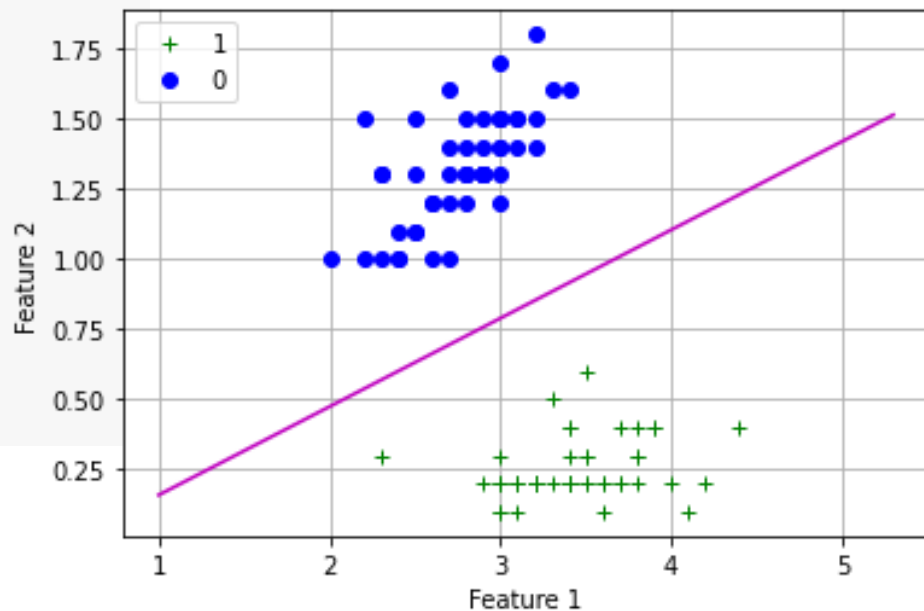
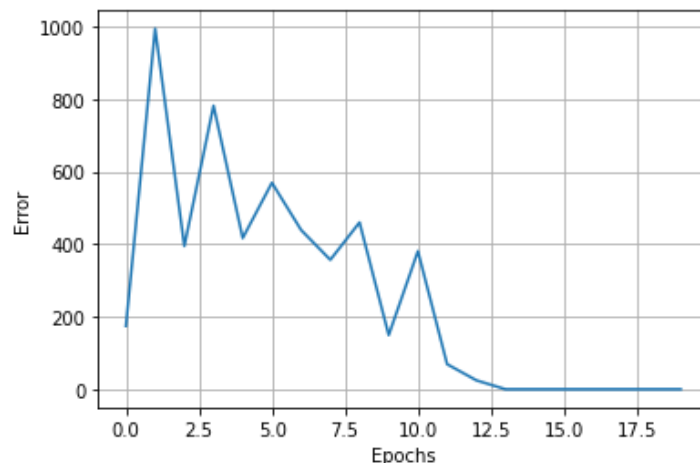
    def fit(self, x,y):
        np.random.RandomState(self.random_state)
        self.w_ = np.random.normal(loc = 0.0, scale = 1.0, size = x.shape[1]+1)
        print(self.w_)
        self.cost_ = []
        for i in range(self.epochs):
            net = np.dot(x,self.w_[1:])+self.w_[0]
            #print(net)
            #print(net.shape)
            output = 1./(1.+np.exp(-net))
            #print(output)
            error = (y-output)
            #print(np.dot(x.T,error))
            #print(x.T.dot(error))
            self.w_[1:]+=self.eta*np.dot(x.T,error)
            self.w_[0]+=self.eta*error.sum()
            #print(y.shape)
            #print(output.shape)
            #print(np.log(output))
            cost = error.mean()#
            cost=-np.dot(y, np.log(output))-np.dot((1-y), np.log(1-output))
            self.cost_.append(cost)
        return(self)

    def predict(self, x):
        net = np.dot(x,self.w_[1:])+self.w_[0]
        output = 1./(1.+np.exp(-net))
        return(np.where(output>=0.5,1,0))
```

```
model = logistic_regression(epochs=20, eta = 0.05)
model.fit(x,y)
y_pred = model.predict(x)
error=np.sum(np.abs(y-y_pred))
print(error)
# decision line
xx1 = np.arange(x[:,0].min()-1,x[:,0].max()+1,0.1)
xx2 = -(model.w_[1]*xx1+model.w_[0])/model.w_[2]
```

```
plt.plot(x[y_pred==0,0],x[y_pred==0,1], 'g+', label='1')
plt.plot(x[y_pred==1,0],x[y_pred==1,1], 'bo', label='0')
plt.legend(loc='upper left')
plt.plot(xx1, xx2, 'm-')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid()
plt.show()

plt.plot(range(len(model.cost_)), model.cost_)
plt.xlabel('Epochs')
plt.ylabel('Error')
plt.grid()
plt.show()
```



Confusion matrix

- A confusion matrix is a table that is often used to **describe the performance of a classification model** (or "classifier") on a set of test data for which the true values are known.

		Predicted Class		
		Positive	Negative	
Actual Class	Positive	True Positive (TP)	False Negative (FN) Type II Error	Sensitivity $\frac{TP}{(TP + FN)}$
	Negative	False Positive (FP) Type I Error	True Negative (TN)	Specificity $\frac{TN}{(TN + FP)}$
		Precision $\frac{TP}{(TP + FP)}$	Negative Predictive Value $\frac{TN}{(TN + FN)}$	Accuracy $\frac{TP + TN}{(TP + TN + FP + FN)}$

Basic terms of a confusion matrix (CM)

- True positives (TP): These are cases in which we predicted yes (they have the disease), and they do have the disease.
- True negatives (TN): We predicted no, and they don't have the disease.
- False positives (FP): We predicted yes, but they don't have the disease. (Also known as a "Type I error")
- False negatives (FN): We predicted no, but they do have the disease. (Also known as a "Type II error")
- Prediction error (ERR): the sum of all false predictions divided by the total number of predictions.
- Accuracy (ACC): the sum of correct (true) predictions divided by the total number of predictions.

- True positive rate (TPR)

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

- False positive rate (FPR)

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

- Precision (PRE):

- Recall (REC):

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}, \quad TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

$$PRE = \frac{TP}{TP + FP}, \quad REC = \frac{TP}{FN + TP}$$

Confusion matrix example

- 1) A binary classifier for medical testing to determine if a patient has certain disease or not (e.g. tumor diagnosis).

	Predicted: NO	Predicted: YES	
Actual: NO	50	10	60
Actual: YES	5	100	105
	55	110	

- 2) There are two possible predicted classes: "yes" and "no"
- 3) If we were predicting the presence of a disease, "**yes**" would mean a patient has the disease, and "**no**" would mean he/she doesn't have the disease.
- 4) The classifier made a total of **165** predictions (e.g., 165 patients were being tested for the presence of that disease).
- 5) Out of those 165 cases, the classifier predicted "**yes**" **110** times, and "**no**" **55** times.
- 6) In reality, 105 patients in the sample have the disease, and 60 patients do not.

	Predicted: NO	Predicted: YES	
n=165			
Actual: NO	50	10	60
Actual: YES	5	100	105
	55	110	

- **Accuracy:** Overall, how often is the classifier correct?
 - $(TP+TN)/total = (100+50)/165 = 0.91$
- **Misclassification Rate:** Overall, how often is it wrong?
 - $(FP+FN)/total = (10+5)/165 = 0.09$
 - equivalent to 1 minus Accuracy
 - also known as "Error Rate"
- **True Positive Rate:** When it's actually yes, how often does it predict yes?
 - $TP/actual\ yes = 100/105 = 0.95$
 - also known as "Sensitivity" or "Recall"
- **False Positive Rate:** When it's actually no, how often does it predict yes?
 - $FP/actual\ no = 10/60 = 0.17$
- **True Negative Rate:** When it's actually no, how often does it predict no?
 - $TN/actual\ no = 50/60 = 0.83$
 - equivalent to 1 minus False Positive Rate
 - also known as "Specificity"
- **Precision:** When it predicts yes, how often is it correct?
 - $TP/predicted\ yes = 100/110 = 0.91$
- **Prevalence:** How often does the yes condition actually occur in our sample?
 - $actual\ yes/total = 105/165 = 0.64$

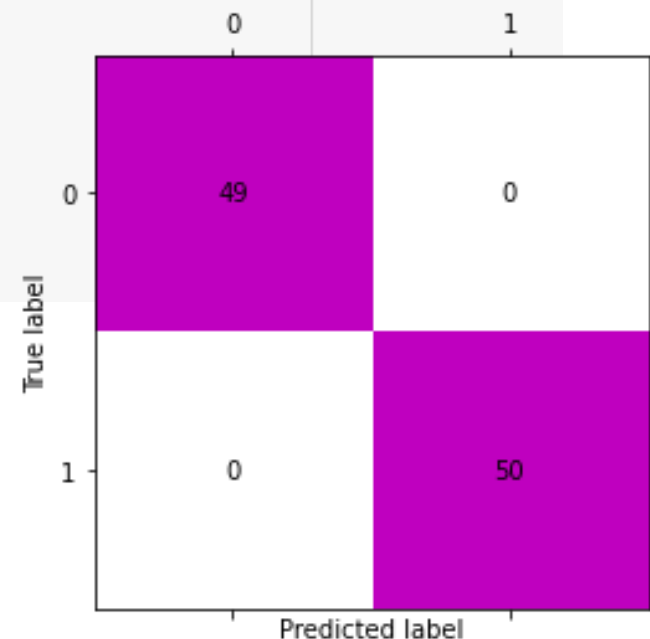
CM for iris

```
from matplotlib.colors import ListedColormap

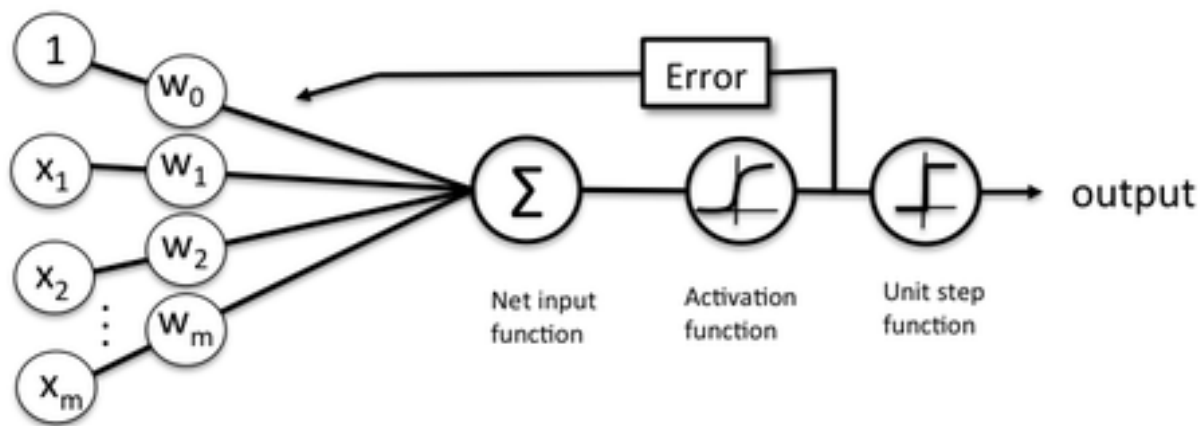
cmap = ListedColormap(['W', 'Y', 'M'])
# Compute confusion matrix
y_true = y
class_names = iris.target_names
print(class_names[0:-1])

#cnf_matrix = confusion_matrix(y_true, y_pred, labels=[0, 1]) # 0 = 'setosa', 1 = 'versicolor'
cm = confusion_matrix(y, y_pred) # y_true and y_pred - 0 = 'setosa', 1 = 'versicolor'
print(cm)

plt.figure()
plt.matshow(cm, cmap=cmap)
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(x=j, y=i, s=cm[i,j], va='center', ha='center')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.show()
```



Recap: single-layer neural network



Schematic of a logistic regression classifier.

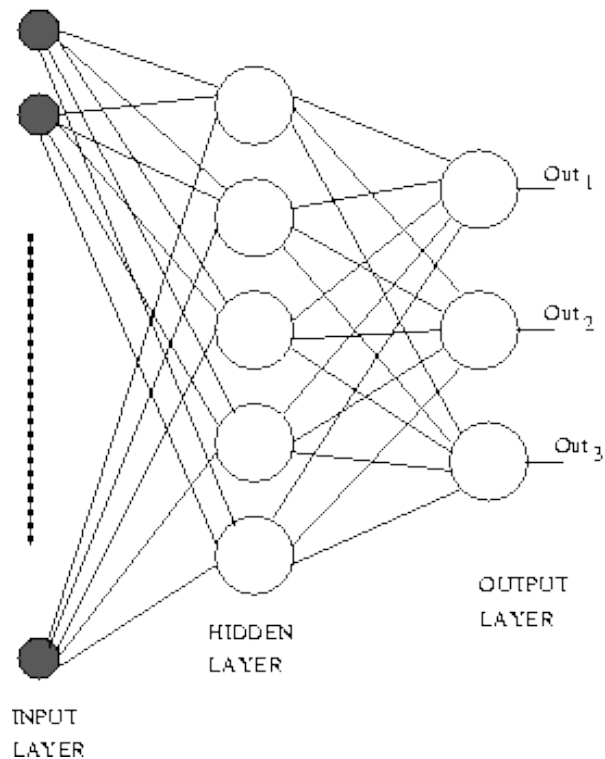
$$J(w) = \frac{1}{2} \sum_{i=1}^m (y(i) - \phi(w^T x_j^{(i)}))^2$$

$$w_j = w_j + \Delta w_j$$

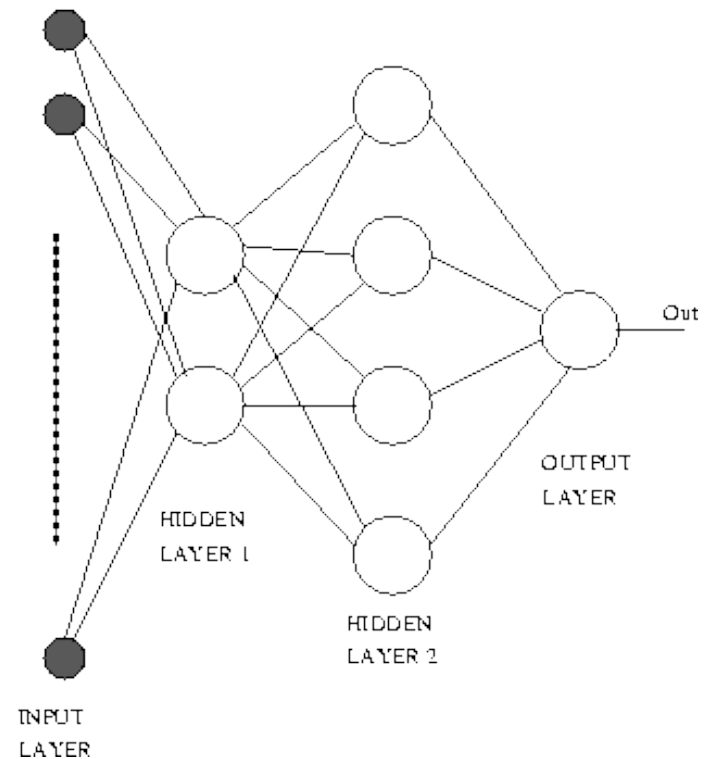
$$\Delta w_j = -\eta \frac{\partial J(w)}{\partial w_j} = \frac{1}{m} \sum_{i=1}^m (y(i) - \phi(w^T x_j^{(i)})) x_j^{(i)}$$

Multilayer neural network

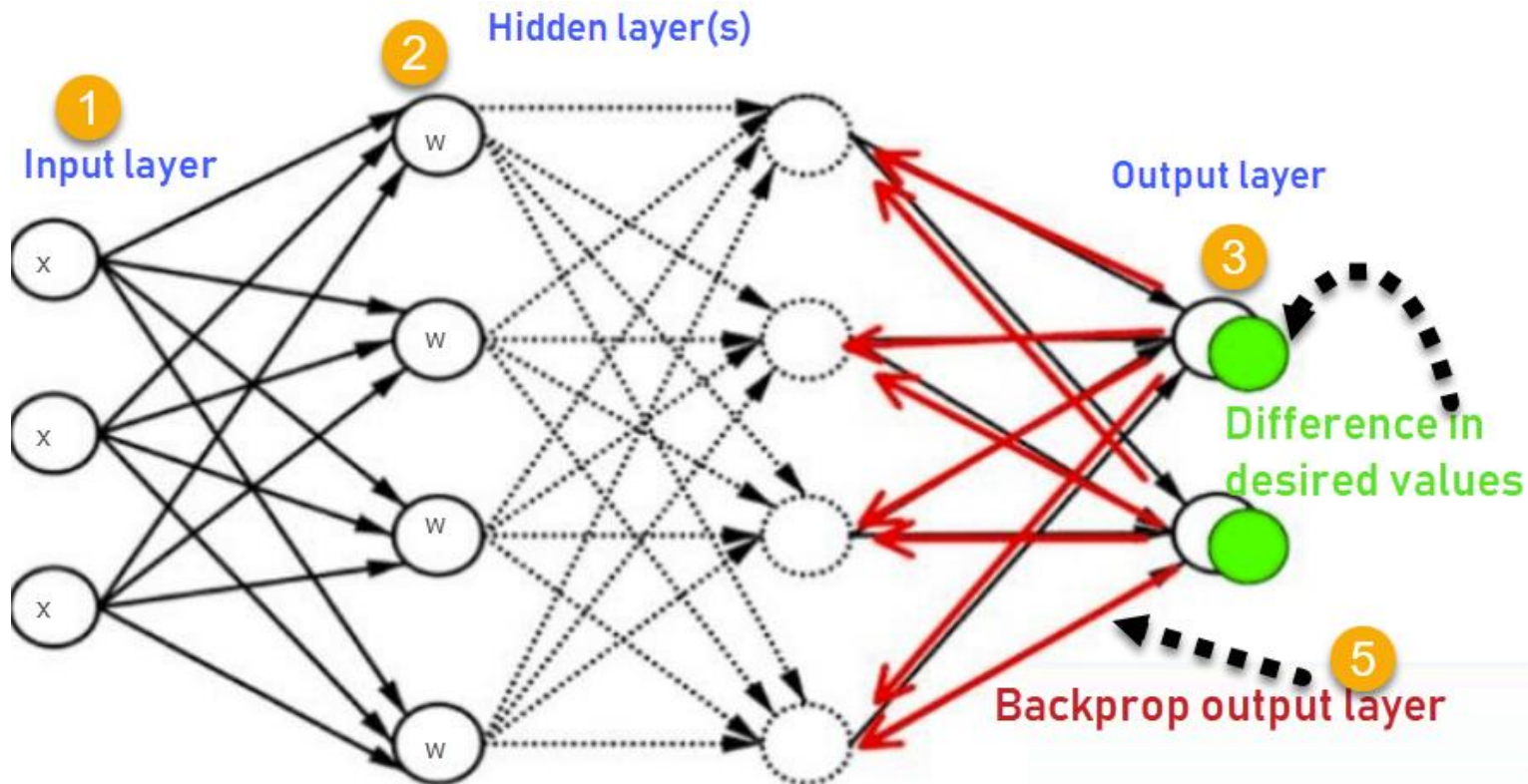
Two-Layer-Network



Three-Layer-Network



How backpropagation works?



MLP in sklearn

```
# MLP
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from matplotlib.colors import ListedColormap
```

```
iris = load_iris()
x = iris.data
y = iris.target
print(iris.target_names)

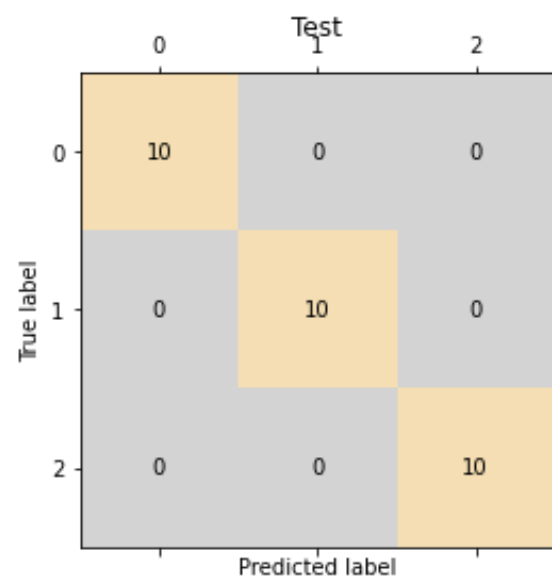
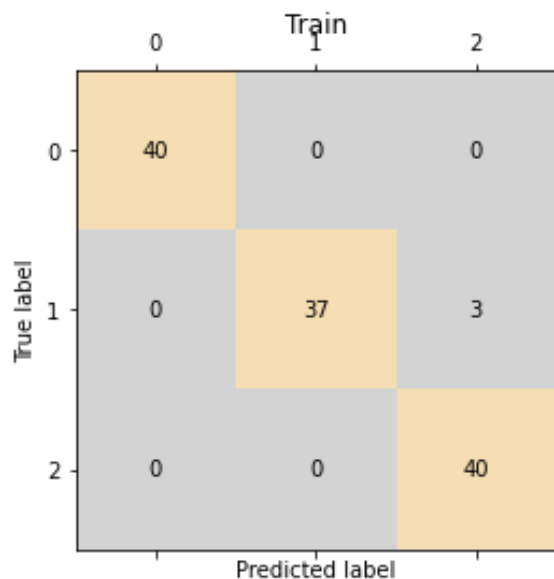
x_train, x_test, y_train, y_test = train_test_split(x, y, stratify=y, random_state=1, test_size=0.2)
np.bincount(y), np.bincount(y_train), np.bincount(y_test)
```

```
mlp =MLPClassifier(solver='sgd', random_state=0, hidden_layer_sizes=[5], alpha = 0.5,max_iter=1000)
# optimizer = 'lbfgs': quasi-Newton methods, 'sgd': stochastic gradient descent, and 'adam': stochastic gradient-based optimizer
mlp.fit(x_train, y_train)
y_pred = mlp.predict(x_test)
print(np.sum(y_test!=y_pred)/len(y_test)*100)

#confusion matrix
def cm(y_test, y_pred):
    cm = confusion_matrix(y_test, y_pred)
    cmap = ListedColormap(['lightgrey', 'silver', 'ghostwhite', 'lavender', 'wheat'])
    cm = confusion_matrix(y_test, y_pred)

    plt.figure()
    plt.matshow(cm, cmap=cmap)
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(x=j, y=i, s=cm[i,j], va='center', ha='center')
    plt.xlabel('Predicted label')
    plt.ylabel('True label')
    plt.show()

cm(y_test, mlp.predict(x_test))
cm(y_train, mlp.predict(x_train))
```



MLP for MNIST dataset

- The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples.
- The digits have been size-normalized and centered in a fixed-size image.
- It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.
- Each digit is 28x28 pixels gray image.





MLP Classifier

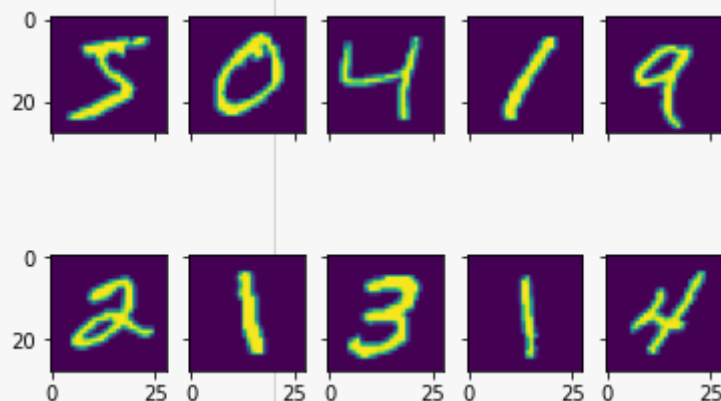
```
# MLP for MNIST
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.utils import to_categorical
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from matplotlib.colors import ListedColormap
```

```
(x_train, y_train), (x_test, y_test)=mnist.load_data() #60000 samples for training, 10000 samples for testing
```

```
print('train:', x_train.shape, y_train.shape)
print('Test set:', x_test.shape, y_test.shape)
```

```
# visualize some
fig, ax = plt.subplots(2,5,sharex=True,sharey=True)
ax = ax.flatten()
for i in range(10):
    ax[i].imshow(x_train[i,:])
plt.show()
```

```
# prepare the data for training
x_train = x_train.reshape(60000,28*28)
x_train = x_train.astype('float32')/255
x_test = x_test.reshape(10000,28*28)
x_test = x_test.astype('float32')/255
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```





```
mlp = MLPClassifier(solver='sgd', activation='logistic', alpha=1e-4, hidden_layer_sizes=(100, 30),
                    random_state=1, max_iter=100, verbose=True, learning_rate_init=.1, tol=1e-4)
mlp.fit(x_train, y_train)

print("Training set score: %f" % mlp.score(x_train, y_train))
print("Test set score: %f" % mlp.score(x_test, y_test))

y_pred = mlp.predict(x_test)

cmap = ListedColormap(['lightgrey', 'silver', 'ghostwhite', 'lavender', 'wheat'])

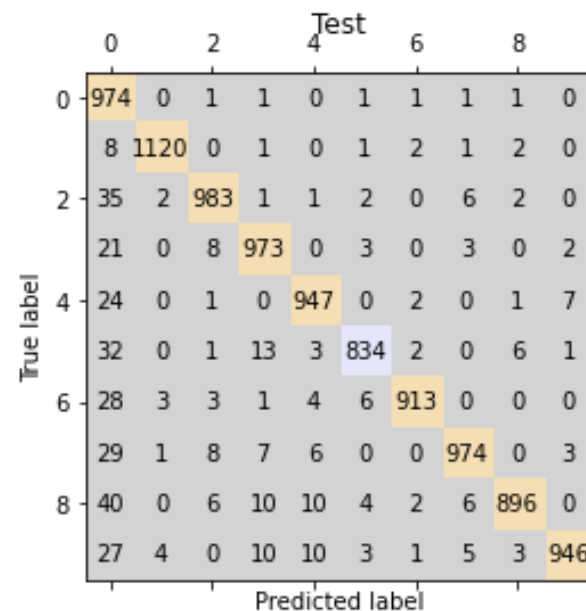
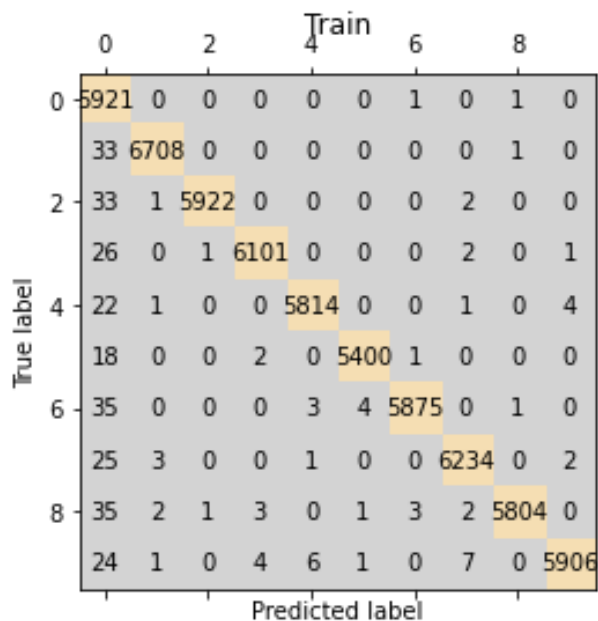
#confusion matrix
def cm(y_test, y_pred, title):
    cm = confusion_matrix(y_test, y_pred)

    plt.figure()
    plt.matshow(cm, cmap=cmap)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(x=j, y=i, s=cm[i,j], va='center', ha='center')

    plt.title(title)
    plt.xlabel('Predicted label')
    plt.ylabel('True label')
    plt.show()

cm(y_train.argmax(1), mlp.predict(x_train).argmax(1), title='Train')
cm(y_test.argmax(1), mlp.predict(x_test).argmax(1), title='Test')
```



	Training	Testing
Accuracy	99.31	94.84

MNIST using Keras

```
# MLP for MNIST
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from sklearn.metrics import confusion_matrix
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 350)	274750
dense_1 (Dense)	(None, 50)	17550
dense_2 (Dense)	(None, 10)	510
Total params: 292,810		
Trainable params: 292,810		
Non-trainable params: 0		

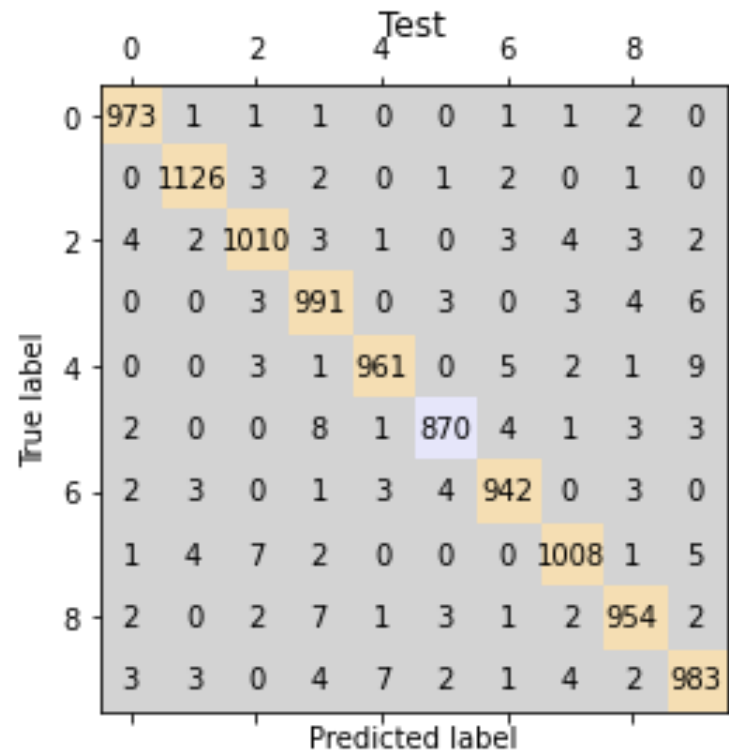
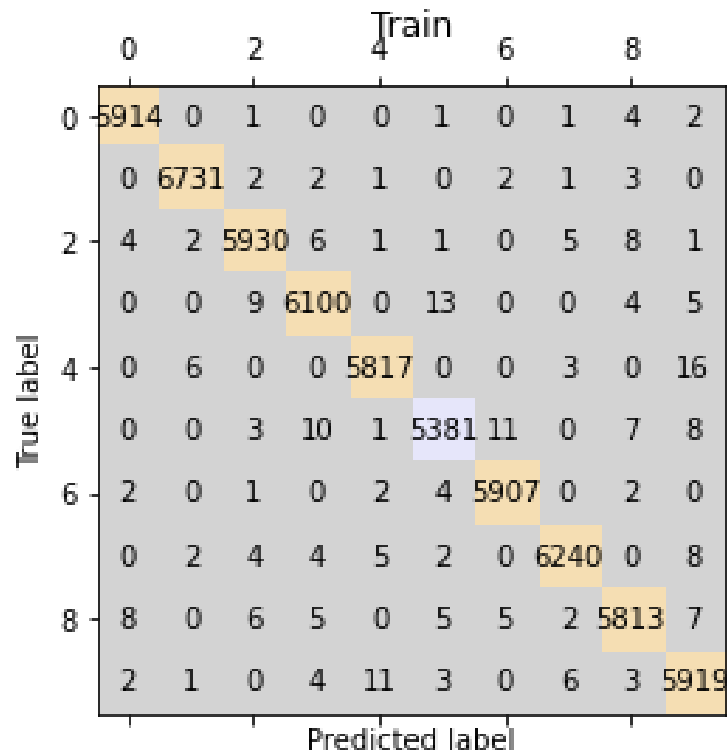
```
# classification model
input_shape=x_train.shape[1]
num_classes = y_train.shape[1]
model = Sequential()
model.add(Dense(350, activation='relu', input_shape=(28*28,)))
model.add(Dense(50, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()

model.fit(x_train, y_train, epochs=100, batch_size=512, verbose=1, validation_split=0.2)

# Test the model after training
test_results = model.evaluate(x_test, y_test, verbose=1)

print(f'Test results \n - Loss      : {test_results[0]: 0.3f} \n - Accuracy: {test_results[1]: 0.3f}')
```



	Loss	Accuracy
Categorical cross entropy	0.122	0.982

Saving and loading models

- Models take a long time to train.
- It is required to save trained models, so we do not have to train them every time.