Supervise Learning

آموختن توانایی تصمیم گیری به کامپیوتر ها با استفاده از دیتاها

- **Supervised learning**
- **Unsipervised learning**
- Reinforcement learning

Supervised learning

- Classification

- Regression

Iris Dataset

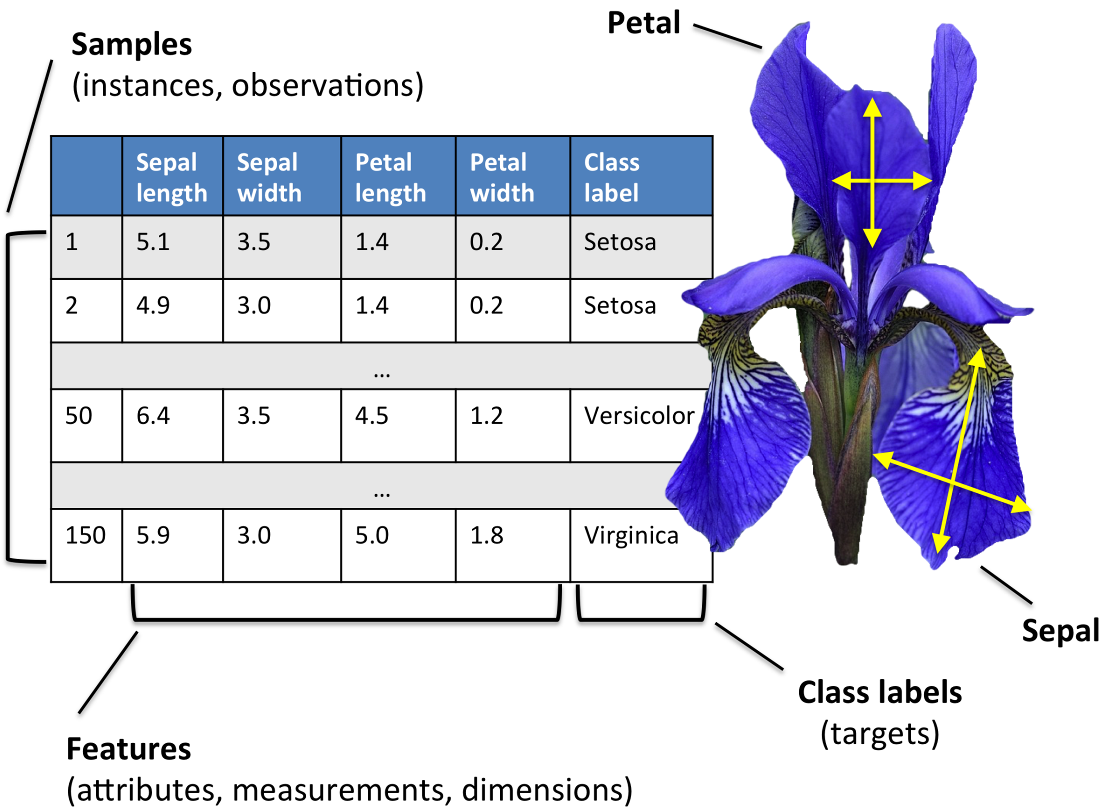


image from http://sebastianraschka.com (http://sebastianraschka.com)

In [5]:
```
from sklearn import datasets
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [6]:
```
iris = datasets.load_iris()
```

In [7]:
```
iris.data.shape
```

Out[7]:
```
(150, 4)
```

In [8]:
```
iris.feature_names
#iris.data
```

Out[8]:
```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

In [9]:
```
iris.target_names
```

Out[9]:
```
array(['setosa', 'versicolor', 'virginica'],
      dtype='<U10')
```

In [10]:
```
iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
iris_df
```

Out[10]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
5	5.4	3.9	1.7	0.4
6	4.6	3.4	1.4	0.3
7	5.0	3.4	1.5	0.2
8	4.4	2.9	1.4	0.2
9	4.9	3.1	1.5	0.1
10	5.4	3.7	1.5	0.2
11	4.8	3.4	1.6	0.2
12	4.8	3.0	1.4	0.1
13	4.3	3.0	1.1	0.1
14	5.8	4.0	1.2	0.2
15	5.7	4.4	1.5	0.4
16	5.4	3.9	1.3	0.4
17	5.1	3.5	1.4	0.3
18	5.7	3.8	1.7	0.3
19	5.1	3.8	1.5	0.3
20	5.4	3.4	1.7	0.2
21	5.1	3.7	1.5	0.4
22	4.6	3.6	1.0	0.2
23	5.1	3.3	1.7	0.5
24	4.8	3.4	1.9	0.2
25	5.0	3.0	1.6	0.2
26	5.0	3.4	1.6	0.4
27	5.2	3.5	1.5	0.2
28	5.2	3.4	1.4	0.2
29	4.7	3.2	1.6	0.2
...	...	...	...	...
120	6.9	3.2	5.7	2.3
121	5.6	2.8	4.9	2.0
122	7.7	2.8	6.7	2.0
123	6.3	2.7	4.9	1.8
124	6.7	3.3	5.7	2.1
125	7.2	3.2	6.0	1.8
126	6.2	2.8	4.8	1.8
127	6.1	3.0	4.9	1.8
128	6.4	2.8	5.6	2.1
129	7.2	3.0	5.8	1.6
130	7.4	2.8	6.1	1.9

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
131	7.9	3.8	6.4	2.0
132	6.4	2.8	5.6	2.2
133	6.3	2.8	5.1	1.5
134	6.1	2.6	5.6	1.4
135	7.7	3.0	6.1	2.3
136	6.3	3.4	5.6	2.4
137	6.4	3.1	5.5	1.8
138	6.0	3.0	4.8	1.8
139	6.9	3.1	5.4	2.1
140	6.7	3.1	5.6	2.4
141	6.9	3.1	5.1	2.3
142	5.8	2.7	5.1	1.9
143	6.8	3.2	5.9	2.3
144	6.7	3.3	5.7	2.5
145	6.7	3.0	5.2	2.3
146	6.3	2.5	5.0	1.9
147	6.5	3.0	5.2	2.0
148	6.2	3.4	5.4	2.3
149	5.9	3.0	5.1	1.8

150 rows × 4 columns

In [11]:
```
iris_df['target'] = iris.target
iris_df
```

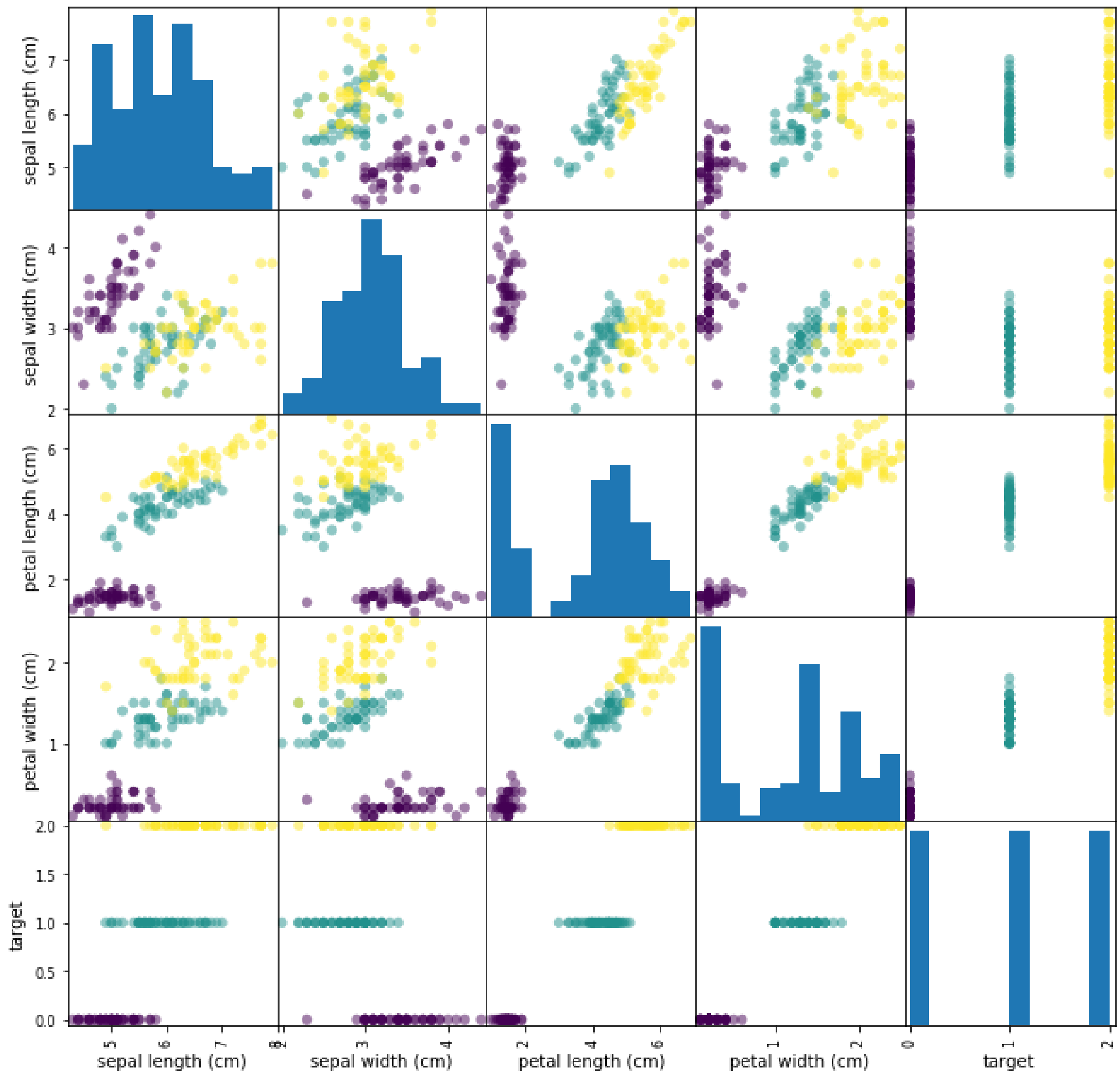
Out[11]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
5	5.4	3.9	1.7	0.4	0
6	4.6	3.4	1.4	0.3	0
7	5.0	3.4	1.5	0.2	0
8	4.4	2.9	1.4	0.2	0
9	4.9	3.1	1.5	0.1	0
10	5.4	3.7	1.5	0.2	0
11	4.8	3.4	1.6	0.2	0
12	4.8	3.0	1.4	0.1	0
13	4.3	3.0	1.1	0.1	0
14	5.8	4.0	1.2	0.2	0
15	5.7	4.4	1.5	0.4	0
16	5.4	3.9	1.3	0.4	0
17	5.1	3.5	1.4	0.3	0
18	5.7	3.8	1.7	0.3	0
19	5.1	3.8	1.5	0.3	0
20	5.4	3.4	1.7	0.2	0
21	5.1	3.7	1.5	0.4	0
22	4.6	3.6	1.0	0.2	0
23	5.1	3.3	1.7	0.5	0
24	4.8	3.4	1.9	0.2	0
25	5.0	3.0	1.6	0.2	0
26	5.0	3.4	1.6	0.4	0
27	5.2	3.5	1.5	0.2	0
28	5.2	3.4	1.4	0.2	0
29	4.7	3.2	1.6	0.2	0
...	...	...	...	...	...
120	6.9	3.2	5.7	2.3	2
121	5.6	2.8	4.9	2.0	2
122	7.7	2.8	6.7	2.0	2
123	6.3	2.7	4.9	1.8	2
124	6.7	3.3	5.7	2.1	2
125	7.2	3.2	6.0	1.8	2
126	6.2	2.8	4.8	1.8	2
127	6.1	3.0	4.9	1.8	2
128	6.4	2.8	5.6	2.1	2
129	7.2	3.0	5.8	1.6	2
130	7.4	2.8	6.1	1.9	2

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
131	7.9	3.8	6.4	2.0	2
132	6.4	2.8	5.6	2.2	2
133	6.3	2.8	5.1	1.5	2
134	6.1	2.6	5.6	1.4	2
135	7.7	3.0	6.1	2.3	2
136	6.3	3.4	5.6	2.4	2
137	6.4	3.1	5.5	1.8	2
138	6.0	3.0	4.8	1.8	2
139	6.9	3.1	5.4	2.1	2
140	6.7	3.1	5.6	2.4	2
141	6.9	3.1	5.1	2.3	2
142	5.8	2.7	5.1	1.9	2
143	6.8	3.2	5.9	2.3	2
144	6.7	3.3	5.7	2.5	2
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

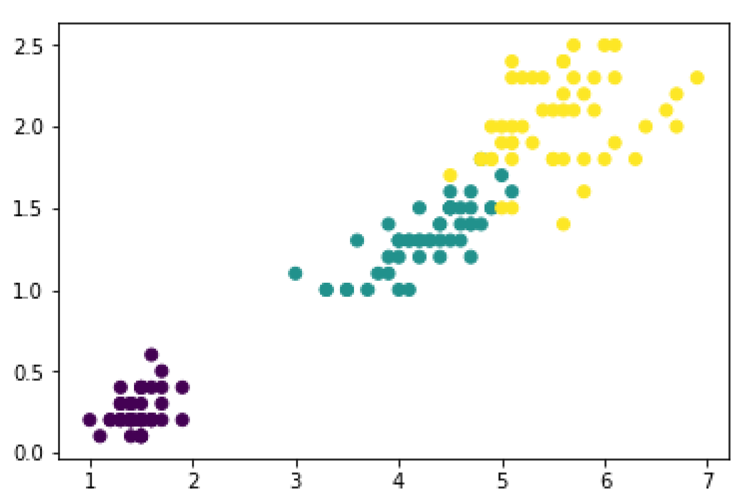
150 rows × 5 columns

In [12]:
```
# Visual EDA
pd.plotting.scatter_matrix(iris_df, c=iris.target, figsize=[11, 11], s=150)
plt.show()
```



KNN : K-Nearest Neighbors

In [13]:
```
from sklearn import datasets

iris = datasets.load_iris()

x = iris.data[:, [2, 3]] #only use petal length and width
y = iris.target

plt.scatter(x[:,0],x[:,1], c=y)
plt.show()
```

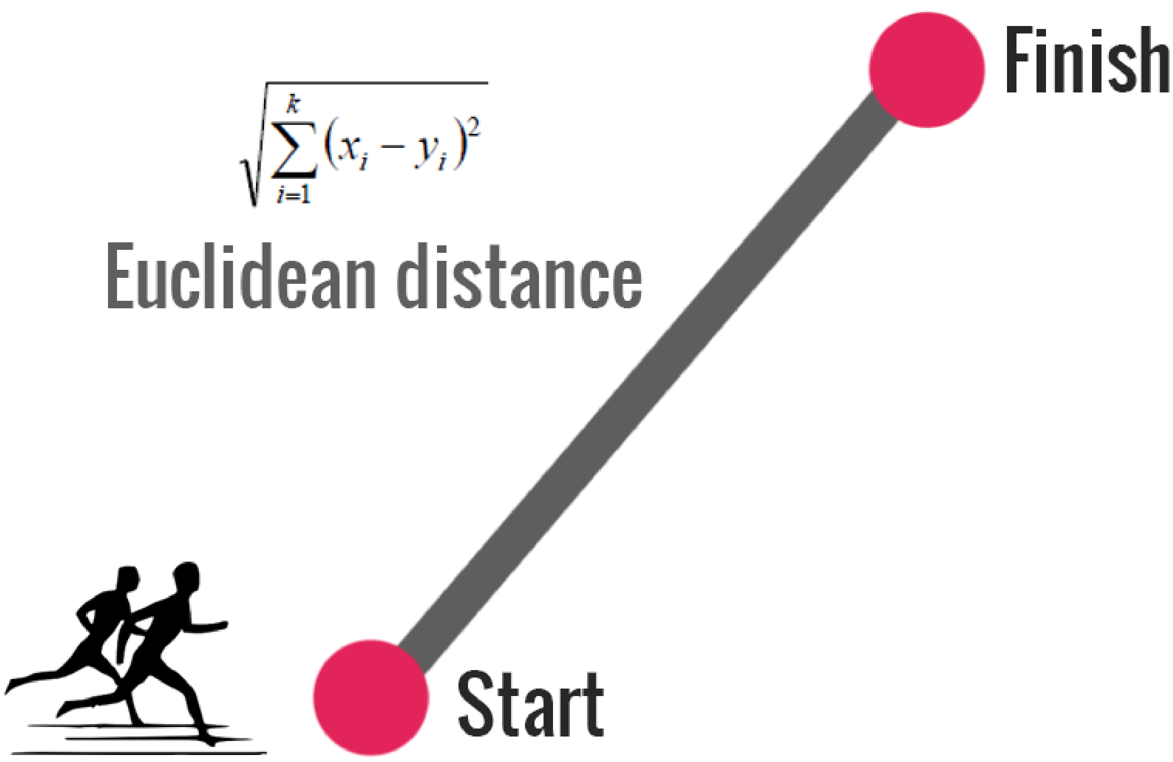


Fit

In [14]:
```python
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=6, metric='minkowski',p=2)

x = iris.data
y = iris.target

knn.fit(x, y)
```

Out[14]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=6, p=2,
           weights='uniform')
```

Distance



ManhattanDistance.png

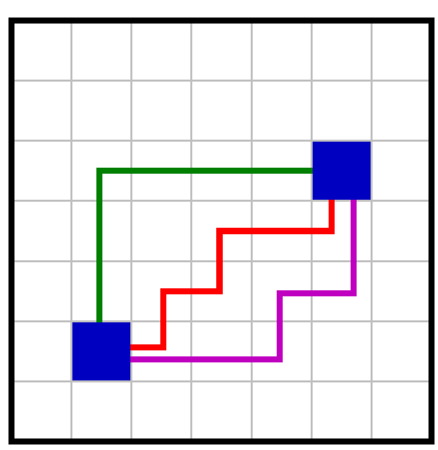
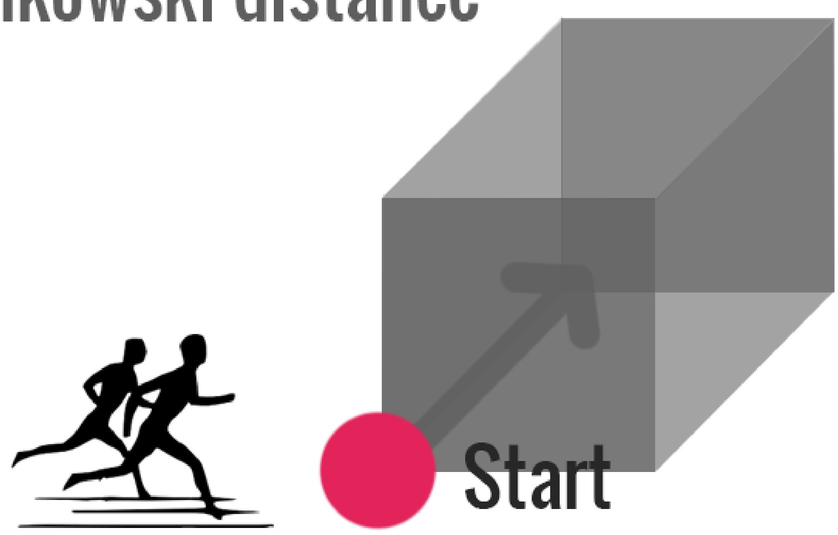


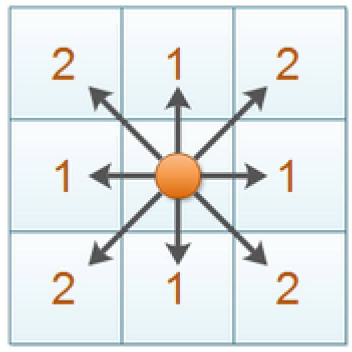
image from https://www.janko.at (https://www.janko.at)

$$d_{(i,j)} = \sqrt[\lambda]{\sum_{k=0}^{n-1} \left| y_{i,k} - y_{j,k} \right|^{\lambda}}$$

Minkowski distance

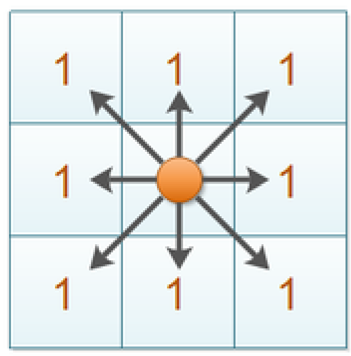


Manhattan Distance



$$|x_1 - x_2| + |y_1 - y_2|$$

Chebyshev Distance



$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

image from : [https://lyfat.wordpress.com (https://lyfat.wordpress.com)](https://lyfat.wordpress.com)

Predict

In [15]:
```
iris.data
```

```
Out[15]: array([[ 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],
                [ 5.4,  3.9,  1.7,  0.4],
                [ 4.6,  3.4,  1.4,  0.3],
                [ 5. ,  3.4,  1.5,  0.2],
                [ 4.4,  2.9,  1.4,  0.2],
                [ 4.9,  3.1,  1.5,  0.1],
                [ 5.4,  3.7,  1.5,  0.2],
                [ 4.8,  3.4,  1.6,  0.2],
                [ 4.8,  3. ,  1.4,  0.1],
                [ 4.3,  3. ,  1.1,  0.1],
                [ 5.8,  4. ,  1.2,  0.2],
                [ 5.7,  4.4,  1.5,  0.4],
                [ 5.4,  3.9,  1.3,  0.4],
                [ 5.1,  3.5,  1.4,  0.3],
                [ 5.7,  3.8,  1.7,  0.3],
                [ 5.1,  3.8,  1.5,  0.3],
                [ 5.4,  3.4,  1.7,  0.2],
                [ 5.1,  3.7,  1.5,  0.4],
                [ 4.6,  3.6,  1. ,  0.2],
                [ 5.1,  3.3,  1.7,  0.5],
                [ 4.8,  3.4,  1.9,  0.2],
                [ 5. ,  3. ,  1.6,  0.2],
                [ 5. ,  3.4,  1.6,  0.4],
                [ 5.2,  3.5,  1.5,  0.2],
                [ 5.2,  3.4,  1.4,  0.2],
                [ 4.7,  3.2,  1.6,  0.2],
                [ 4.8,  3.1,  1.6,  0.2],
                [ 5.4,  3.4,  1.5,  0.4],
                [ 5.2,  4.1,  1.5,  0.1],
                [ 5.5,  4.2,  1.4,  0.2],
                [ 4.9,  3.1,  1.5,  0.1],
                [ 5. ,  3.2,  1.2,  0.2],
                [ 5.5,  3.5,  1.3,  0.2],
                [ 4.9,  3.1,  1.5,  0.1],
                [ 4.4,  3. ,  1.3,  0.2],
                [ 5.1,  3.4,  1.5,  0.2],
                [ 5. ,  3.5,  1.3,  0.3],
                [ 4.5,  2.3,  1.3,  0.3],
                [ 4.4,  3.2,  1.3,  0.2],
                [ 5. ,  3.5,  1.6,  0.6],
                [ 5.1,  3.8,  1.9,  0.4],
                [ 4.8,  3. ,  1.4,  0.3],
                [ 5.1,  3.8,  1.6,  0.2],
                [ 4.6,  3.2,  1.4,  0.2],
                [ 5.3,  3.7,  1.5,  0.2],
                [ 5. ,  3.3,  1.4,  0.2],
                [ 7. ,  3.2,  4.7,  1.4],
                [ 6.4,  3.2,  4.5,  1.5],
                [ 6.9,  3.1,  4.9,  1.5],
                [ 5.5,  2.3,  4. ,  1.3],
                [ 6.5,  2.8,  4.6,  1.5],
                [ 5.7,  2.8,  4.5,  1.3],
                [ 6.3,  3.3,  4.7,  1.6],
                [ 4.9,  2.4,  3.3,  1. ],
                [ 6.6,  2.9,  4.6,  1.3],
                [ 5.2,  2.7,  3.9,  1.4],
                [ 5. ,  2. ,  3.5,  1. ],
                [ 5.9,  3. ,  4.2,  1.5],
                [ 6. ,  2.2,  4. ,  1. ],
                [ 6.1,  2.9,  4.7,  1.4],
                [ 5.6,  2.9,  3.6,  1.3],
                [ 6.7,  3.1,  4.4,  1.4],
                [ 5.6,  3. ,  4.5,  1.5],
                [ 5.8,  2.7,  4.1,  1. ],
                [ 6.2,  2.2,  4.5,  1.5],
                [ 5.6,  2.5,  3.9,  1.1],
                [ 5.9,  3.2,  4.8,  1.8],
                [ 6.1,  2.8,  4. ,  1.3],
                [ 6.3,  2.5,  4.9,  1.5],
                [ 6.1,  2.8,  4.7,  1.2],
```

```
[ 6.4,  2.9,  4.3,  1.3],
[ 6.6,  3. ,  4.4,  1.4],
[ 6.8,  2.8,  4.8,  1.4],
[ 6.7,  3. ,  5. ,  1.7],
[ 6. ,  2.9,  4.5,  1.5],
[ 5.7,  2.6,  3.5,  1. ],
[ 5.5,  2.4,  3.8,  1.1],
[ 5.5,  2.4,  3.7,  1. ],
[ 5.8,  2.7,  3.9,  1.2],
[ 6. ,  2.7,  5.1,  1.6],
[ 5.4,  3. ,  4.5,  1.5],
[ 6. ,  3.4,  4.5,  1.6],
[ 6.7,  3.1,  4.7,  1.5],
[ 6.3,  2.3,  4.4,  1.3],
[ 5.6,  3. ,  4.1,  1.3],
[ 5.5,  2.5,  4. ,  1.3],
[ 5.5,  2.6,  4.4,  1.2],
[ 6.1,  3. ,  4.6,  1.4],
[ 5.8,  2.6,  4. ,  1.2],
[ 5. ,  2.3,  3.3,  1. ],
[ 5.6,  2.7,  4.2,  1.3],
[ 5.7,  3. ,  4.2,  1.2],
[ 5.7,  2.9,  4.2,  1.3],
[ 6.2,  2.9,  4.3,  1.3],
[ 5.1,  2.5,  3. ,  1.1],
[ 5.7,  2.8,  4.1,  1.3],
[ 6.3,  3.3,  6. ,  2.5],
[ 5.8,  2.7,  5.1,  1.9],
[ 7.1,  3. ,  5.9,  2.1],
[ 6.3,  2.9,  5.6,  1.8],
[ 6.5,  3. ,  5.8,  2.2],
[ 7.6,  3. ,  6.6,  2.1],
[ 4.9,  2.5,  4.5,  1.7],
[ 7.3,  2.9,  6.3,  1.8],
[ 6.7,  2.5,  5.8,  1.8],
[ 7.2,  3.6,  6.1,  2.5],
[ 6.5,  3.2,  5.1,  2. ],
[ 6.4,  2.7,  5.3,  1.9],
[ 6.8,  3. ,  5.5,  2.1],
[ 5.7,  2.5,  5. ,  2. ],
[ 5.8,  2.8,  5.1,  2.4],
[ 6.4,  3.2,  5.3,  2.3],
[ 6.5,  3. ,  5.5,  1.8],
[ 7.7,  3.8,  6.7,  2.2],
[ 7.7,  2.6,  6.9,  2.3],
[ 6. ,  2.2,  5. ,  1.5],
[ 6.9,  3.2,  5.7,  2.3],
[ 5.6,  2.8,  4.9,  2. ],
[ 7.7,  2.8,  6.7,  2. ],
[ 6.3,  2.7,  4.9,  1.8],
[ 6.7,  3.3,  5.7,  2.1],
[ 7.2,  3.2,  6. ,  1.8],
[ 6.2,  2.8,  4.8,  1.8],
[ 6.1,  3. ,  4.9,  1.8],
[ 6.4,  2.8,  5.6,  2.1],
[ 7.2,  3. ,  5.8,  1.6],
[ 7.4,  2.8,  6.1,  1.9],
[ 7.9,  3.8,  6.4,  2. ],
[ 6.4,  2.8,  5.6,  2.2],
[ 6.3,  2.8,  5.1,  1.5],
[ 6.1,  2.6,  5.6,  1.4],
[ 7.7,  3. ,  6.1,  2.3],
[ 6.3,  3.4,  5.6,  2.4],
[ 6.4,  3.1,  5.5,  1.8],
[ 6. ,  3. ,  4.8,  1.8],
[ 6.9,  3.1,  5.4,  2.1],
[ 6.7,  3.1,  5.6,  2.4],
[ 6.9,  3.1,  5.1,  2.3],
[ 5.8,  2.7,  5.1,  1.9],
[ 6.8,  3.2,  5.9,  2.3],
[ 6.7,  3.3,  5.7,  2.5],
[ 6.7,  3. ,  5.2,  2.3],
[ 6.3,  2.5,  5. ,  1.9],
[ 6.5,  3. ,  5.2,  2. ],
```

```
        [ 6.2,  3.4,  5.4,  2.3],
        [ 5.9,  3. ,  5.1,  1.8]])
```

In [16]:
```python
xx = np.array([[5, 3, 1, 0.2]])
yy = knn.predict(xx)
print(yy)
```

```
[0]
```

Train and test

In [17]:
```python
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=
0.3, random_state=42, stratify=iris.target)
```

In [18]:
```python
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train, y_train)
y_predict = knn.predict(x_test)
y_predict
```

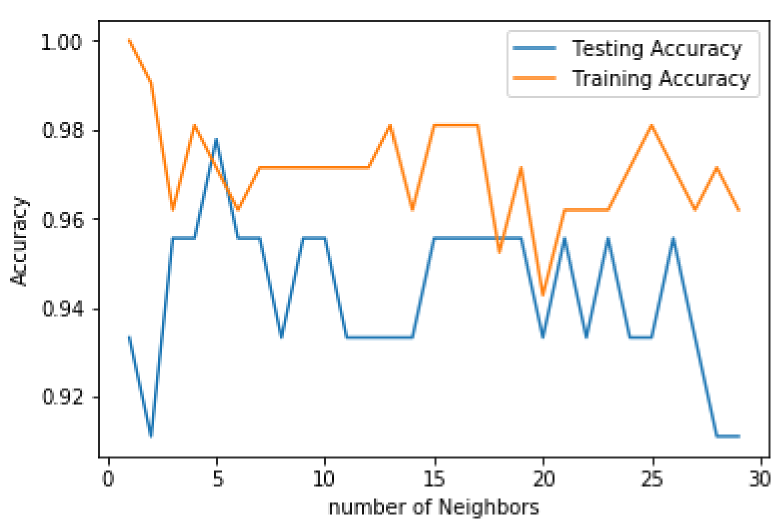
Out[18]:
```
array([2, 1, 2, 1, 2, 2, 1, 1, 0, 2, 0, 0, 2, 2, 0, 2, 1, 0, 0, 0, 1, 0, 1,
       2, 2, 1, 1, 1, 1, 0, 2, 2, 1, 0, 2, 0, 0, 0, 0, 1, 1, 0, 1, 2, 1])
```

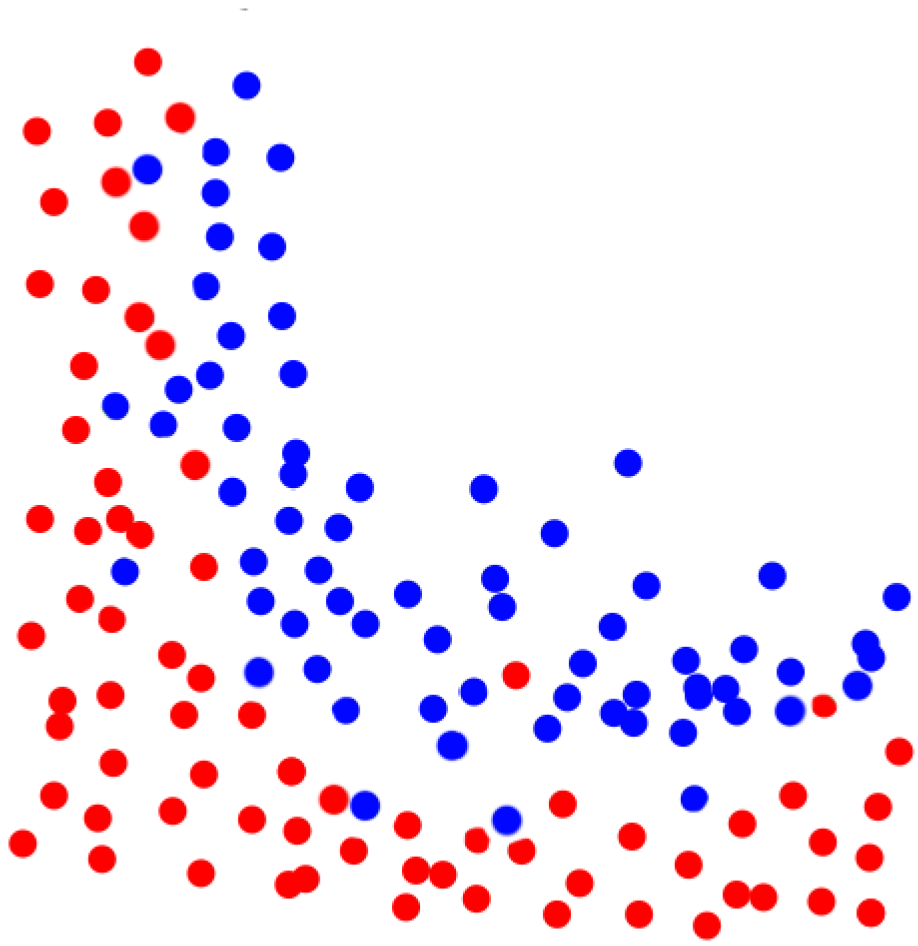
In [19]:
```python
knn.score(x_test, y_test)
```

Out[19]:
```
0.9777777777777775
```

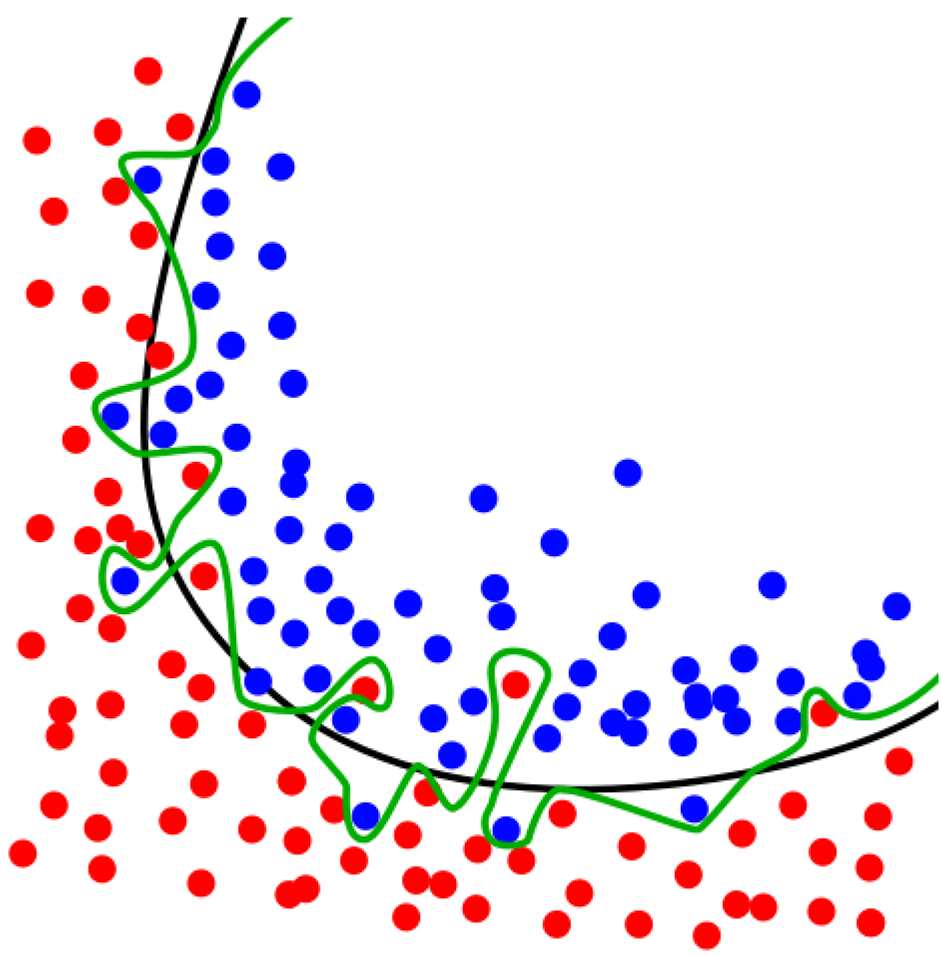
Over Fitting and Under Fitting

In [20]:
```python
neighbors = np.arange(1, 30)
train_accuracy = np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

for i,k in enumerate(neighbors):
    knn_model = KNeighborsClassifier(n_neighbors=k)
    knn_model.fit(x_train, y_train)
    train_accuracy[i] = knn_model.score(x_train, y_train)
    test_accuracy[i] = knn_model.score(x_test, y_test)

plt.plot(neighbors, test_accuracy, label = 'Testing Accuracy')
plt.plot(neighbors, train_accuracy, label = 'Training Accuracy')
plt.legend()
plt.xlabel('number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```

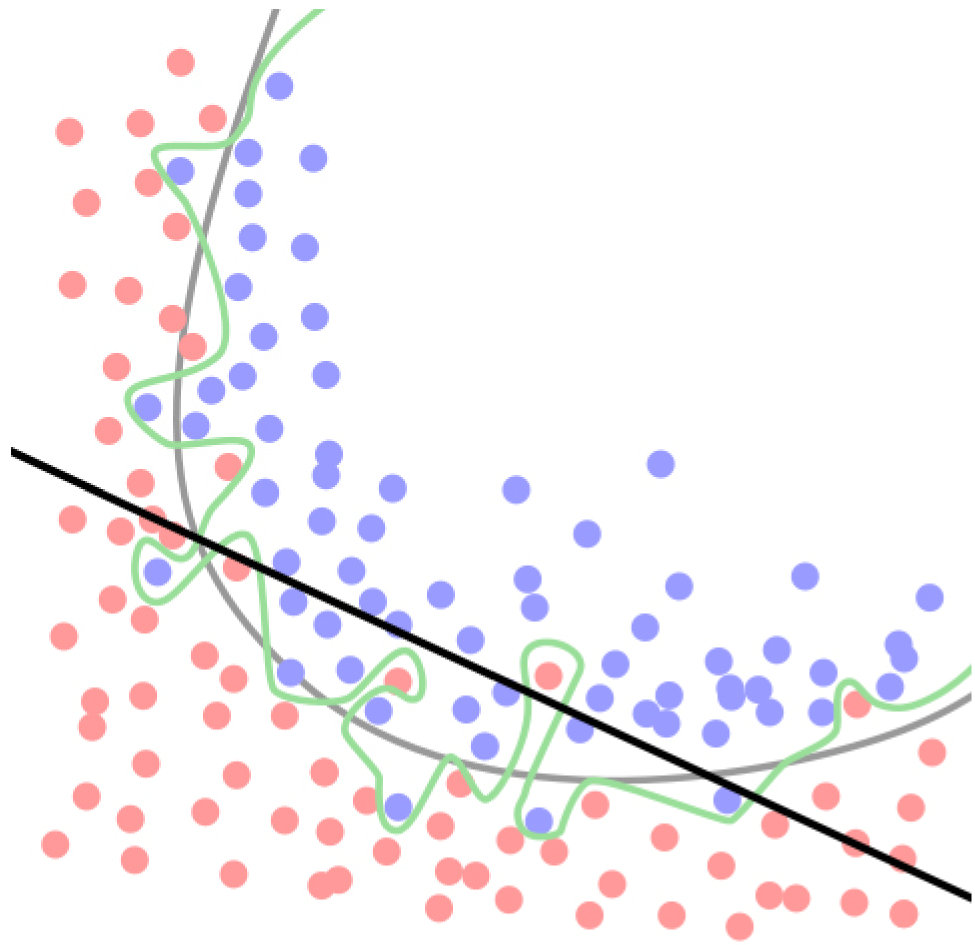




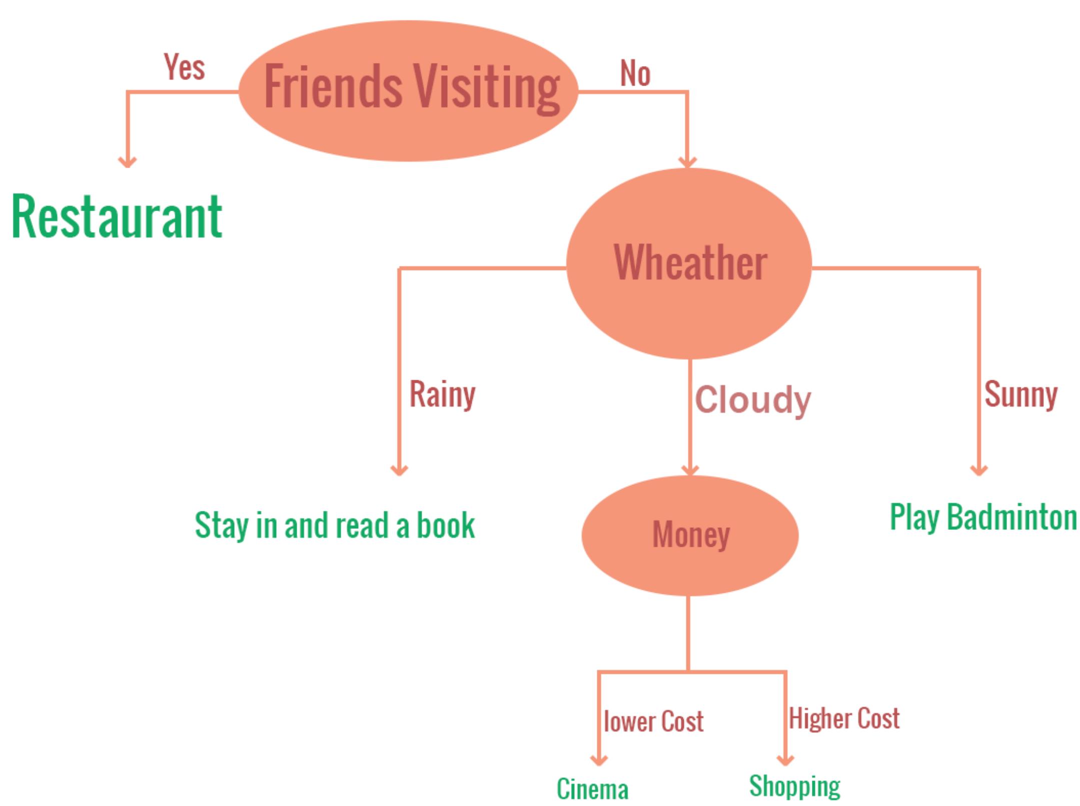
Over Fitting



Under fitting



Decision Tree



iris dataset

In [21]:
```
from sklearn.tree import DecisionTreeClassifier

dtc = DecisionTreeClassifier()
dtc = dtc.fit(x_train, y_train)
```

In [22]:
```
predict_dtc = dtc.predict(x_train[:, :])
```

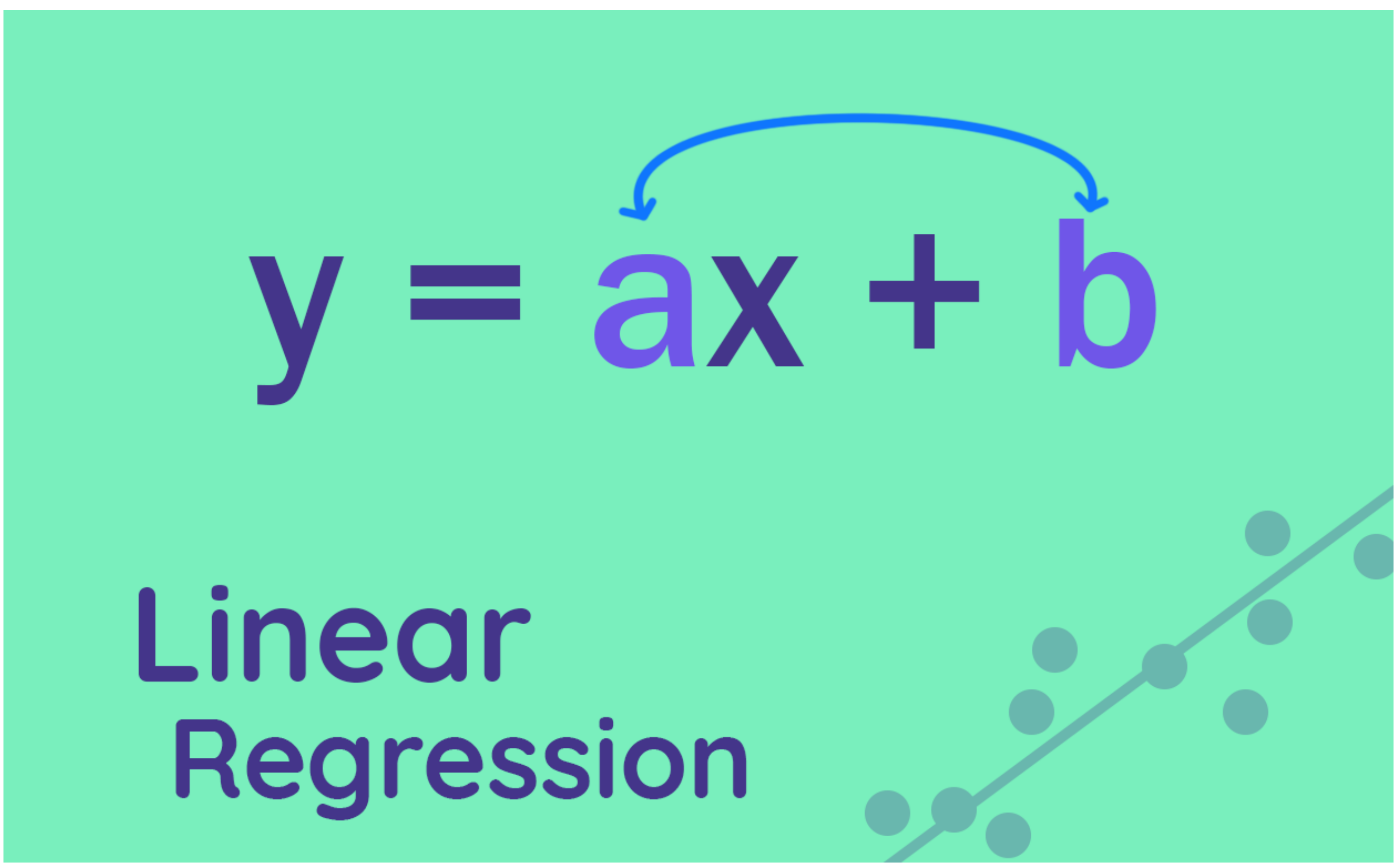
In [23]:
```
from sklearn import metrics
metrics.accuracy_score(y_train, predict_dtc )
```

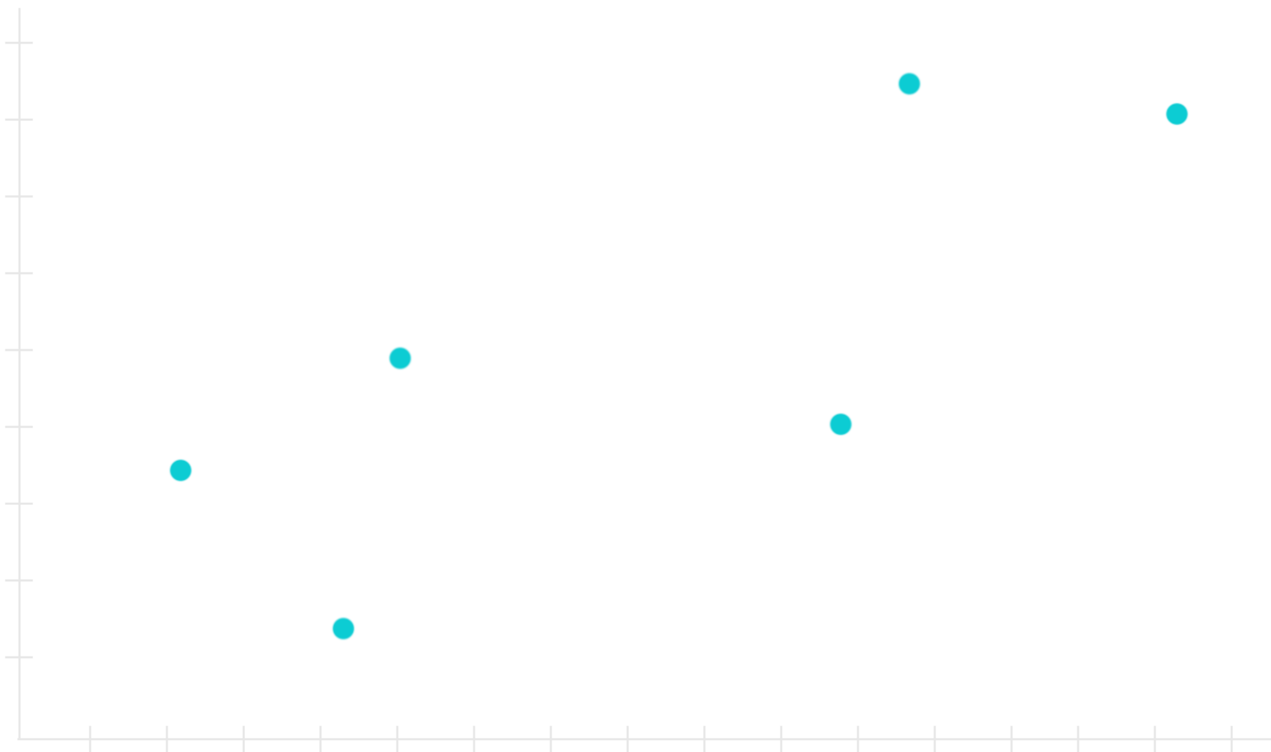
Out[23]: 1.0

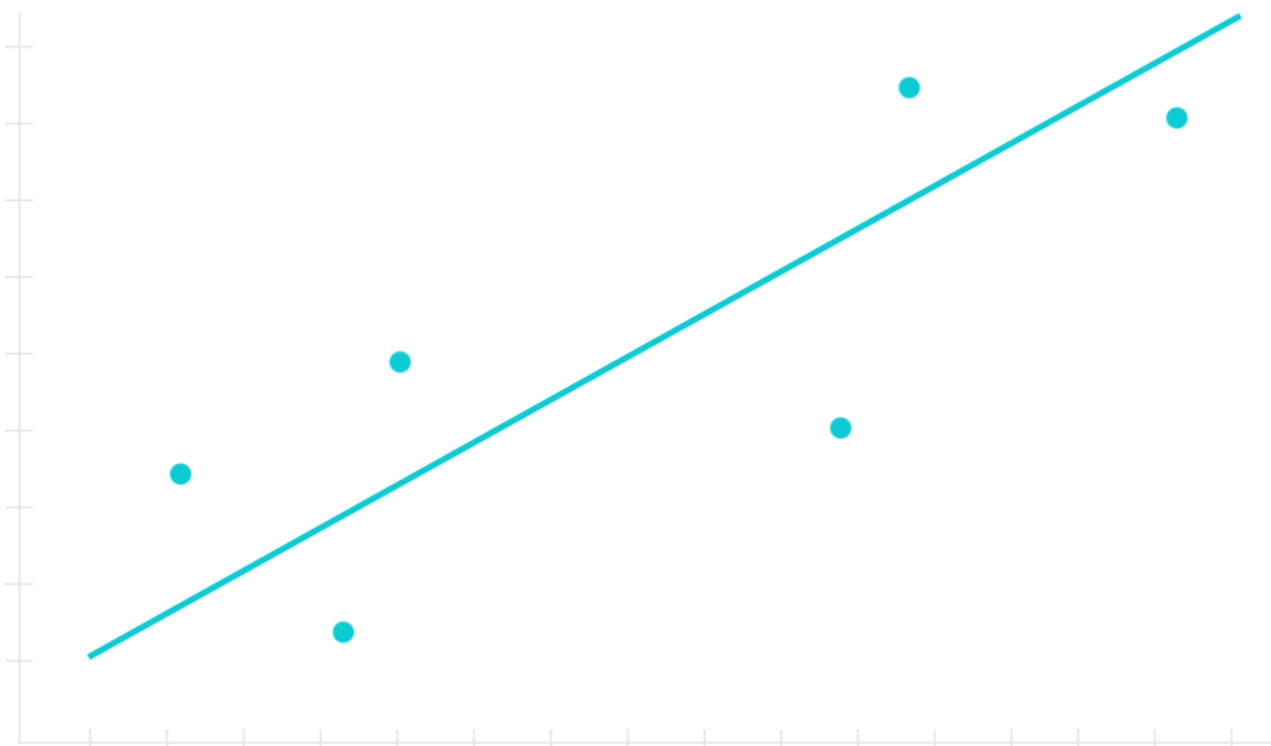
Regression

- Linear Regression
- Logistic Regresion

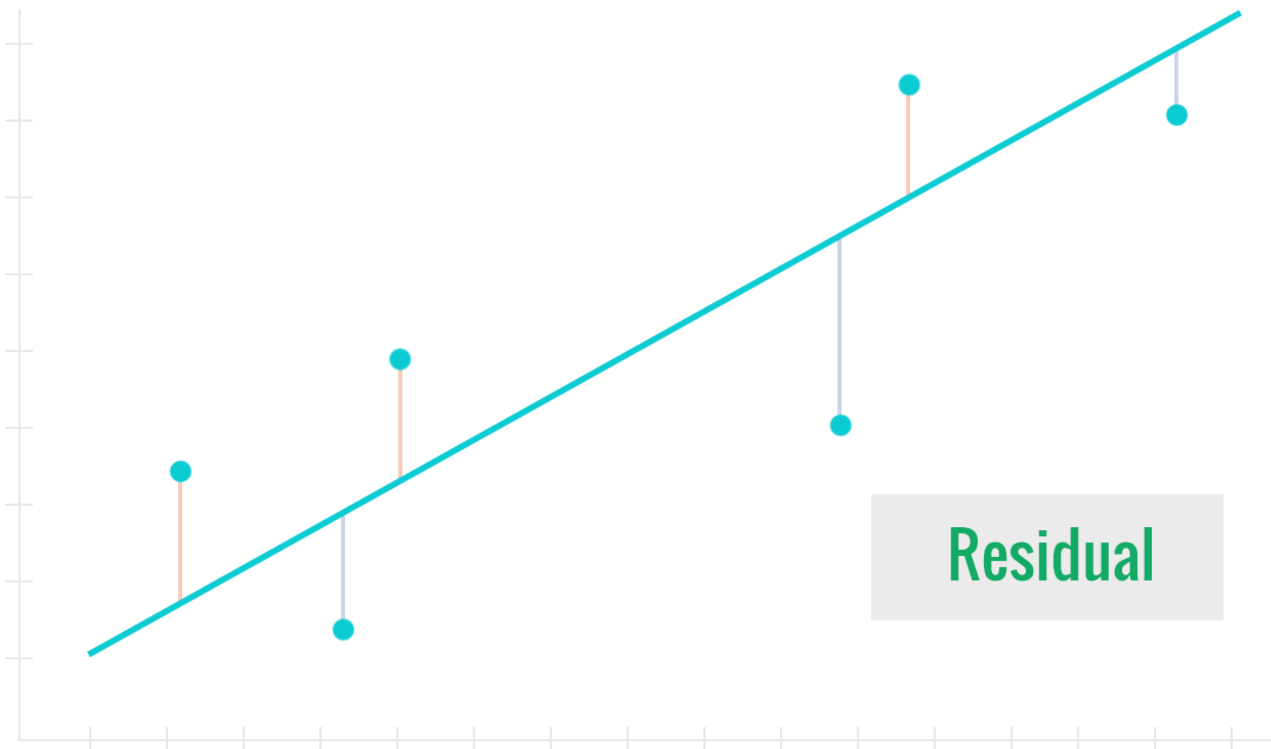
Linear Regression







Residual



Ordinary least squares (OLS) : Minimize sum of square of residuals to building the model

Linear regression in higher dimensions

$$y = a_1x_1 + a_2x_2 + b$$

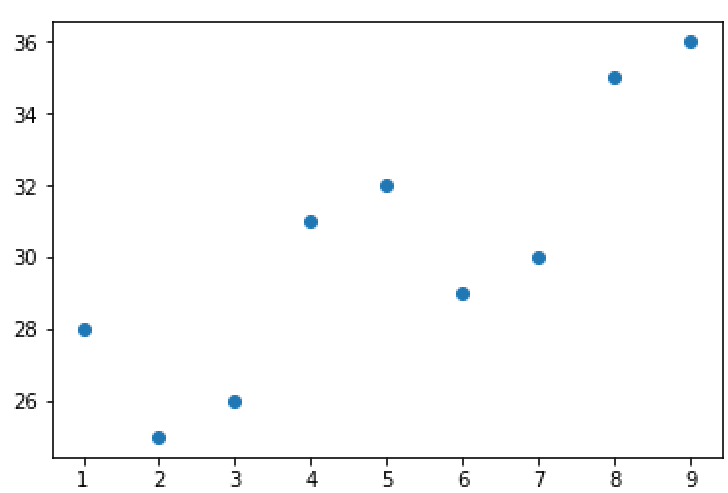
$$y = a_1x_1 + a_2x_2 + \ldots + a_nx_n + b$$

Regression example

In [24]:
```
from sklearn.linear_model import LinearRegression
```

In [25]:
```
x = np.arange(1,10)
y= np.array([28, 25, 26, 31, 32, 29, 30, 35, 36])
```

In [26]:
```
import matplotlib.pyplot as plt
plt.scatter(x,y)
plt.show()
```

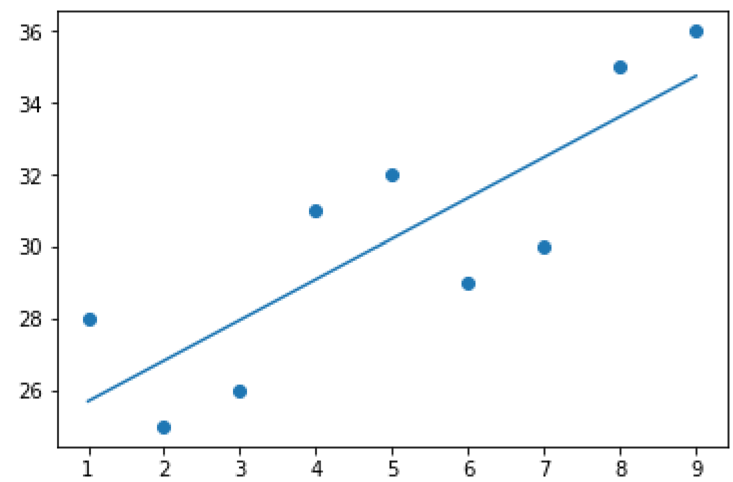


In [27]:
```
x = x.reshape(-1,1)
y = y.reshape(-1,1)
reg = LinearRegression()
reg.fit(x,y)
```

Out[27]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

In [28]:
```
yhat = reg.predict(x)
```

In [29]:
```
plt.scatter(x,y)
plt.plot(x,yhat)
plt.show()
```



In [30]:
```
from sklearn.datasets import load_boston
```

In [31]:
```
boston = load_boston()
boston_df = pd.DataFrame(boston.data, columns=boston.feature_names)
boston_df['Price'] = boston.target
boston_df
```

Out[31]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	39
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	39
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	39
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	39
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	39
5	0.02985	0.0	2.18	0.0	0.458	6.430	58.7	6.0622	3.0	222.0	18.7	39
6	0.08829	12.5	7.87	0.0	0.524	6.012	66.6	5.5605	5.0	311.0	15.2	39
7	0.14455	12.5	7.87	0.0	0.524	6.172	96.1	5.9505	5.0	311.0	15.2	39
8	0.21124	12.5	7.87	0.0	0.524	5.631	100.0	6.0821	5.0	311.0	15.2	38
9	0.17004	12.5	7.87	0.0	0.524	6.004	85.9	6.5921	5.0	311.0	15.2	38
10	0.22489	12.5	7.87	0.0	0.524	6.377	94.3	6.3467	5.0	311.0	15.2	39
11	0.11747	12.5	7.87	0.0	0.524	6.009	82.9	6.2267	5.0	311.0	15.2	39
12	0.09378	12.5	7.87	0.0	0.524	5.889	39.0	5.4509	5.0	311.0	15.2	39
13	0.62976	0.0	8.14	0.0	0.538	5.949	61.8	4.7075	4.0	307.0	21.0	39
14	0.63796	0.0	8.14	0.0	0.538	6.096	84.5	4.4619	4.0	307.0	21.0	38
15	0.62739	0.0	8.14	0.0	0.538	5.834	56.5	4.4986	4.0	307.0	21.0	39
16	1.05393	0.0	8.14	0.0	0.538	5.935	29.3	4.4986	4.0	307.0	21.0	38
17	0.78420	0.0	8.14	0.0	0.538	5.990	81.7	4.2579	4.0	307.0	21.0	38
18	0.80271	0.0	8.14	0.0	0.538	5.456	36.6	3.7965	4.0	307.0	21.0	28
19	0.72580	0.0	8.14	0.0	0.538	5.727	69.5	3.7965	4.0	307.0	21.0	39
20	1.25179	0.0	8.14	0.0	0.538	5.570	98.1	3.7979	4.0	307.0	21.0	37
21	0.85204	0.0	8.14	0.0	0.538	5.965	89.2	4.0123	4.0	307.0	21.0	39
22	1.23247	0.0	8.14	0.0	0.538	6.142	91.7	3.9769	4.0	307.0	21.0	39
23	0.98843	0.0	8.14	0.0	0.538	5.813	100.0	4.0952	4.0	307.0	21.0	39
24	0.75026	0.0	8.14	0.0	0.538	5.924	94.1	4.3996	4.0	307.0	21.0	39
25	0.84054	0.0	8.14	0.0	0.538	5.599	85.7	4.4546	4.0	307.0	21.0	30
26	0.67191	0.0	8.14	0.0	0.538	5.813	90.3	4.6820	4.0	307.0	21.0	37
27	0.95577	0.0	8.14	0.0	0.538	6.047	88.8	4.4534	4.0	307.0	21.0	30
28	0.77299	0.0	8.14	0.0	0.538	6.495	94.4	4.4547	4.0	307.0	21.0	38
29	1.00245	0.0	8.14	0.0	0.538	6.674	87.3	4.2390	4.0	307.0	21.0	38
...	...	...	...	...	...	...	...	...	...	...	...	...
476	4.87141	0.0	18.10	0.0	0.614	6.484	93.6	2.3053	24.0	666.0	20.2	39
477	15.02340	0.0	18.10	0.0	0.614	5.304	97.3	2.1007	24.0	666.0	20.2	34
478	10.23300	0.0	18.10	0.0	0.614	6.185	96.7	2.1705	24.0	666.0	20.2	37
479	14.33370	0.0	18.10	0.0	0.614	6.229	88.0	1.9512	24.0	666.0	20.2	38
480	5.82401	0.0	18.10	0.0	0.532	6.242	64.7	3.4242	24.0	666.0	20.2	39
481	5.70818	0.0	18.10	0.0	0.532	6.750	74.9	3.3317	24.0	666.0	20.2	39
482	5.73116	0.0	18.10	0.0	0.532	7.061	77.0	3.4106	24.0	666.0	20.2	39
483	2.81838	0.0	18.10	0.0	0.532	5.762	40.3	4.0983	24.0	666.0	20.2	39
484	2.37857	0.0	18.10	0.0	0.583	5.871	41.9	3.7240	24.0	666.0	20.2	37
485	3.67367	0.0	18.10	0.0	0.583	6.312	51.9	3.9917	24.0	666.0	20.2	38
486	5.69175	0.0	18.10	0.0	0.583	6.114	79.8	3.5459	24.0	666.0	20.2	39

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
487	4.83567	0.0	18.10	0.0	0.583	5.905	53.2	3.1523	24.0	666.0	20.2	38
488	0.15086	0.0	27.74	0.0	0.609	5.454	92.7	1.8209	4.0	711.0	20.1	39
489	0.18337	0.0	27.74	0.0	0.609	5.414	98.3	1.7554	4.0	711.0	20.1	34
490	0.20746	0.0	27.74	0.0	0.609	5.093	98.0	1.8226	4.0	711.0	20.1	31
491	0.10574	0.0	27.74	0.0	0.609	5.983	98.8	1.8681	4.0	711.0	20.1	39
492	0.11132	0.0	27.74	0.0	0.609	5.983	83.5	2.1099	4.0	711.0	20.1	39
493	0.17331	0.0	9.69	0.0	0.585	5.707	54.0	2.3817	6.0	391.0	19.2	39
494	0.27957	0.0	9.69	0.0	0.585	5.926	42.6	2.3817	6.0	391.0	19.2	39
495	0.17899	0.0	9.69	0.0	0.585	5.670	28.8	2.7986	6.0	391.0	19.2	39
496	0.28960	0.0	9.69	0.0	0.585	5.390	72.9	2.7986	6.0	391.0	19.2	39
497	0.26838	0.0	9.69	0.0	0.585	5.794	70.6	2.8927	6.0	391.0	19.2	39
498	0.23912	0.0	9.69	0.0	0.585	6.019	65.3	2.4091	6.0	391.0	19.2	39
499	0.17783	0.0	9.69	0.0	0.585	5.569	73.5	2.3999	6.0	391.0	19.2	39
500	0.22438	0.0	9.69	0.0	0.585	6.027	79.7	2.4982	6.0	391.0	19.2	39
501	0.06263	0.0	11.93	0.0	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	39
502	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	39
503	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	39
504	0.10959	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	39
505	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	39

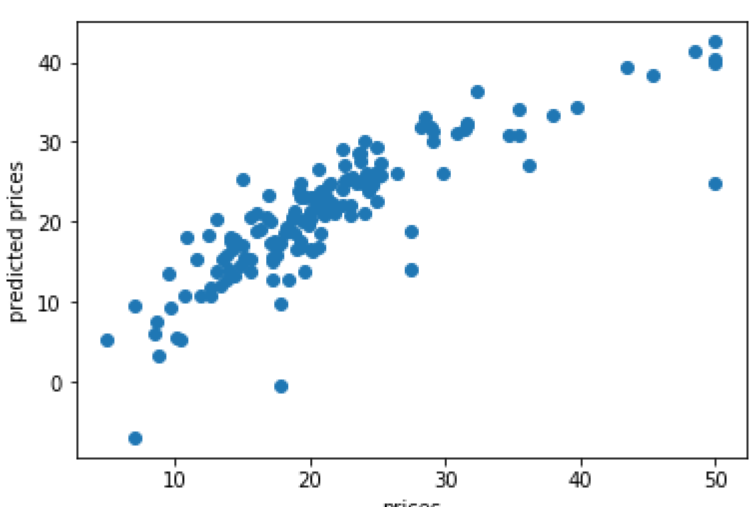
506 rows × 14 columns

In [32]:
```
x = boston.data
y = boston.target
```

In [33]:
```
x_train, x_test ,y_train, y_test  = train_test_split(x, y, test_size = 0.3, random_stat
e=42)
```

In [34]:
```
reg = LinearRegression()
reg.fit(x_train, y_train)
y_pred = reg.predict(x_test)
```

In [35]:
```
plt.scatter(y_test, y_pred)
plt.plot()
plt.xlabel('prices')
plt.ylabel('predicted prices')
plt.show()
```



Mean square error (MSE) : to evaluating the model

In [36]:
```
import sklearn.metrics
mse = metrics.mean_squared_error(y_test, y_pred)
mse
```

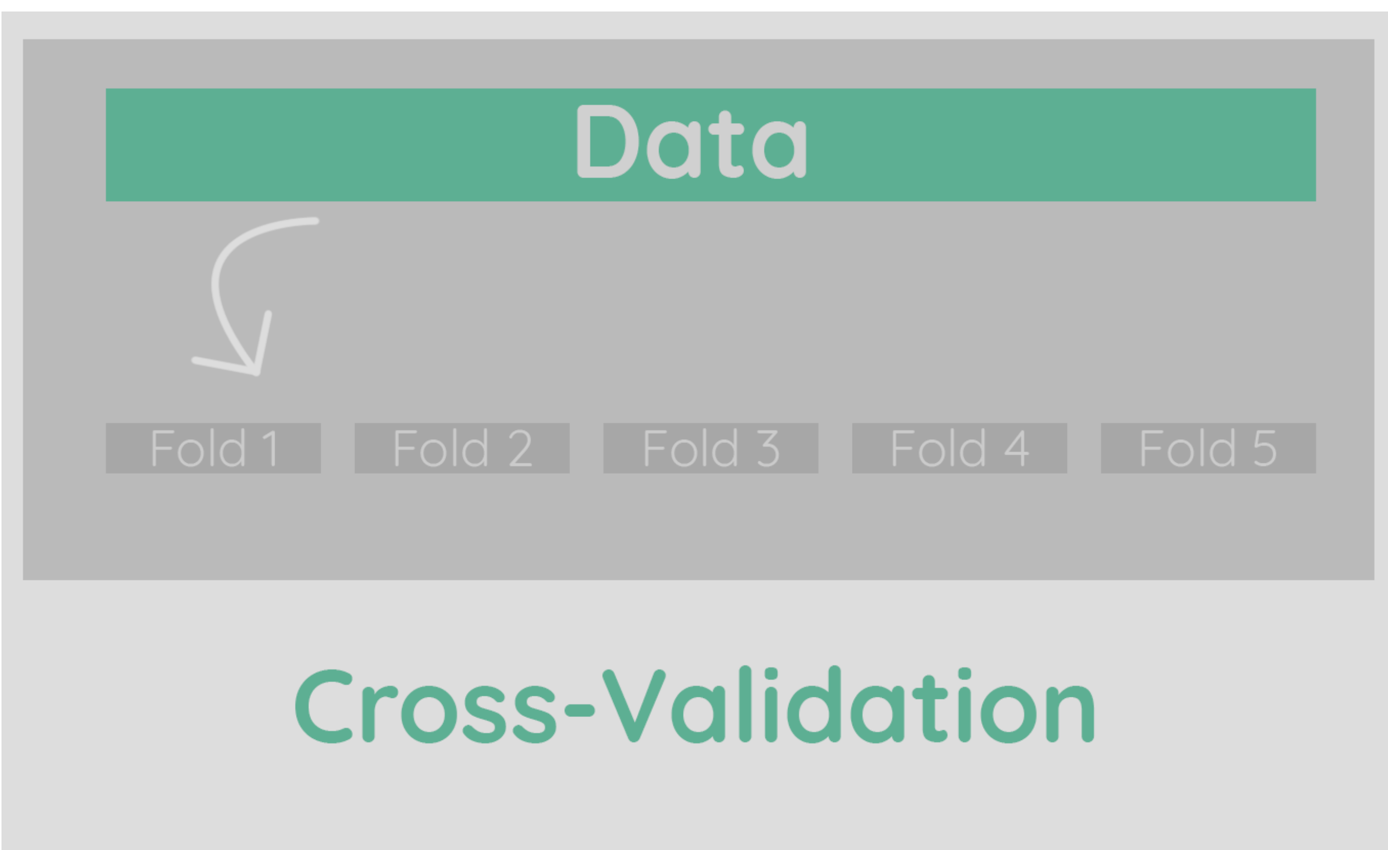
Out[36]: 21.540218943931421

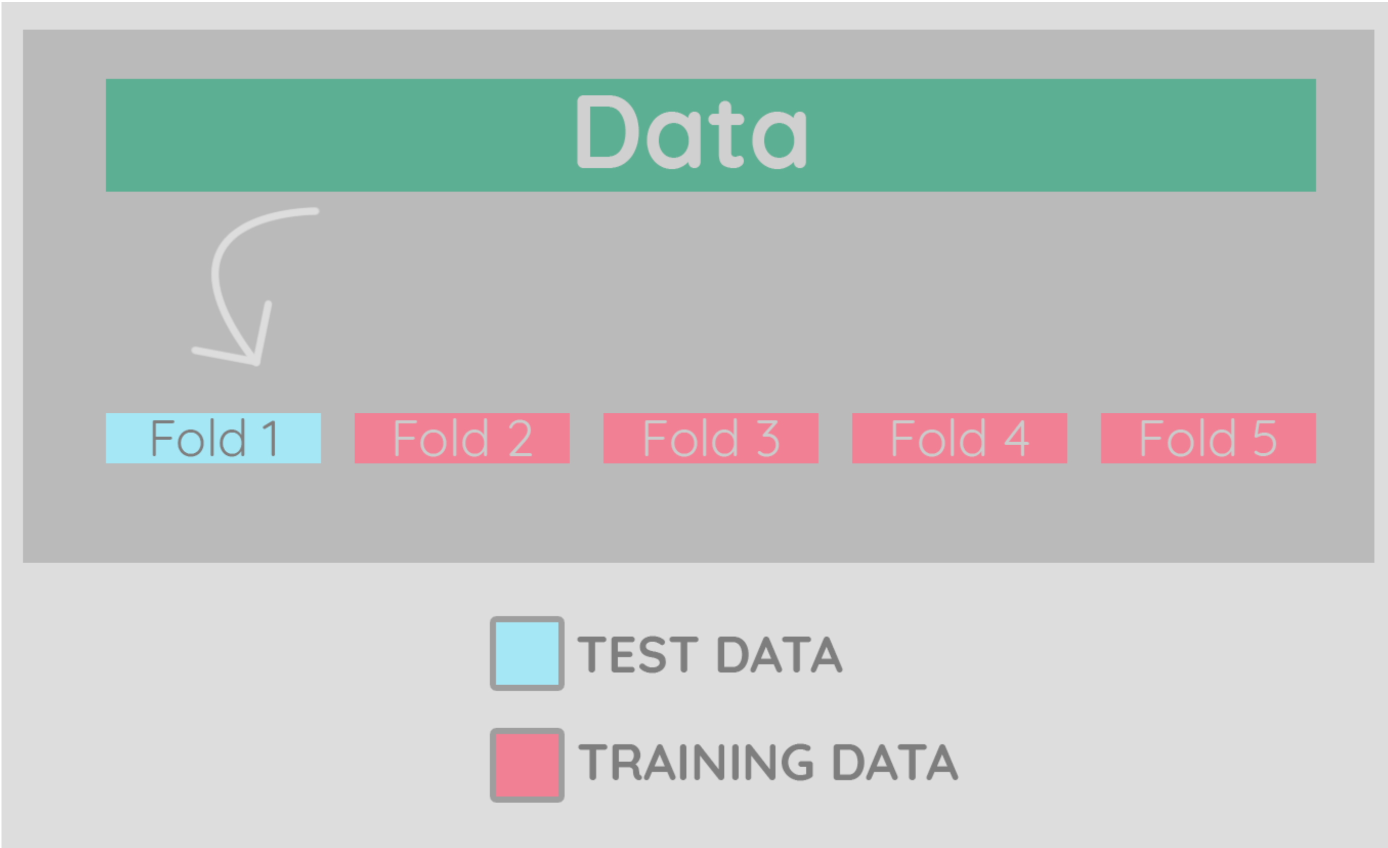
In [37]:
```
new_x = boston.data[:,[1,2]]
new_y = boston.target

new_x_train, new_x_test ,new_y_train, new_y_test  = train_test_split(new_x, new_y, test
_size = 0.3, random_state=42)

new_reg = LinearRegression()
new_reg.fit(new_x_train, new_y_train)
new_y_predict = new_reg.predict(new_x_test)

new_mse = metrics.mean_squared_error(new_y_test, new_y_predict)
new_mse
```

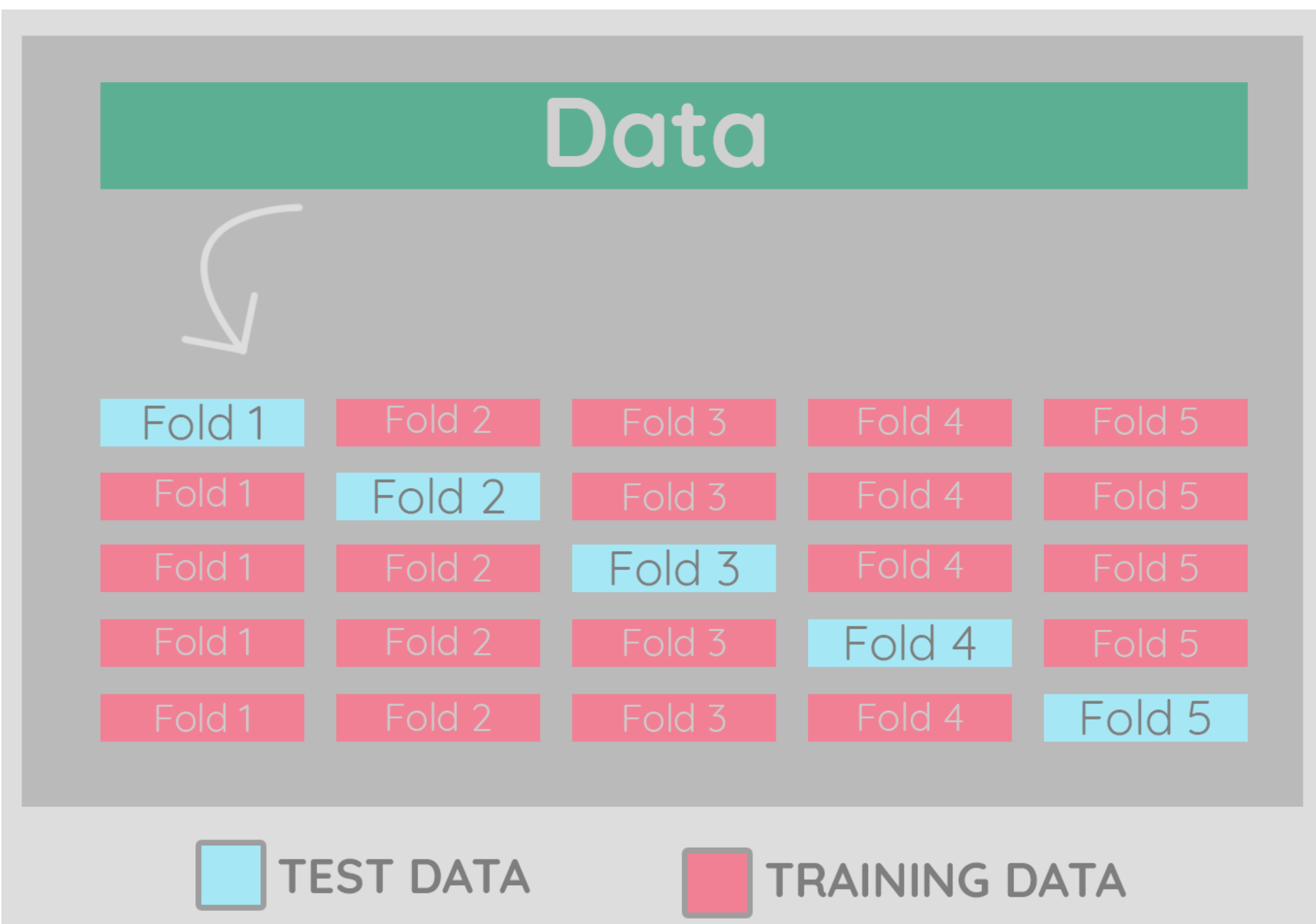
Out[37]: 52.49477133220752

CrossValidation (K-Fold Cross Validation)

> To optimize the results, we can use Cross Validation technique.







In [38]:
```
from sklearn.model_selection import cross_val_score

reg = LinearRegression()
first_cv_scores = cross_val_score(reg, x, y, cv=5)
second_cv_scores = cross_val_score(reg, x, y, cv=10)

print('mean in first_cv_scores is {0:.2f} and in second_cv_scores is {1:.2f}'.format(np
.mean(first_cv_scores), np.mean(second_cv_scores)) )
```

```
mean in first_cv_scores is 0.35 and in second_cv_scores is 0.20
```

$Regularization\ Regression:$

$$CostFunction = OLS + regularization\ term(Penalty)$$

$$Penalizing\ large\ coefficients\ =\ Penalizing\ Overfitting$$

$$ridge\ regression\ lost\ function = OLS_{(ordinary\ Least\ of\ squre)} + \alpha * \sum_{i=1}^{n} {a_i}^2$$

$$Lasso\ regression\ lost\ function = OLS_{(ordinary\ Least\ of\ squre)} + \alpha * \sum_{i=1}^{n} |a_i|$$

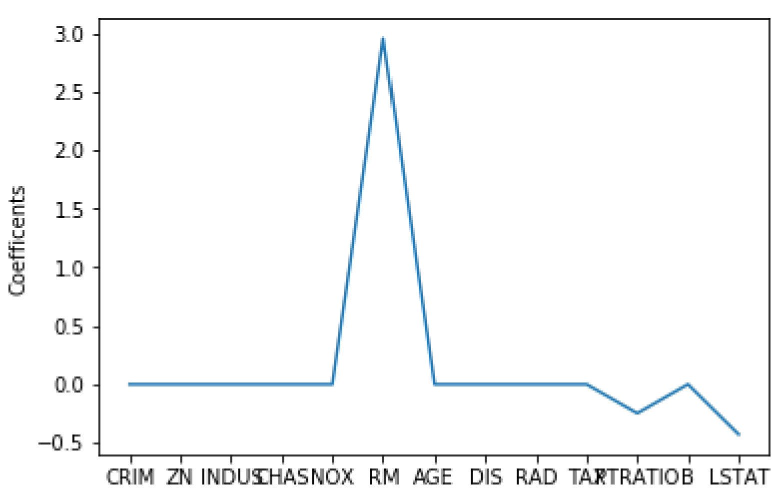
$$\alpha : is\ a\ constant\ we\ predict\ and\ is\ similar\ to\ picking\ k\ in\ KNN. if\ alpha\ equal\ to\ zero\ we\ get\ back\ OLS$$

$$and\ very\ high\ alpha\ can\ lead\ to\ underfitting$$

$$a : coefficents$$

In [39]:
```python
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.1, normalize=True)
lasso.fit(x, y)
lasso_coef = lasso.coef_

print(lasso_coef)

plt.plot(range(13), lasso_coef)
plt.xticks(range(13), boston.feature_names)
plt.ylabel('Coefficents')
plt.show()
```

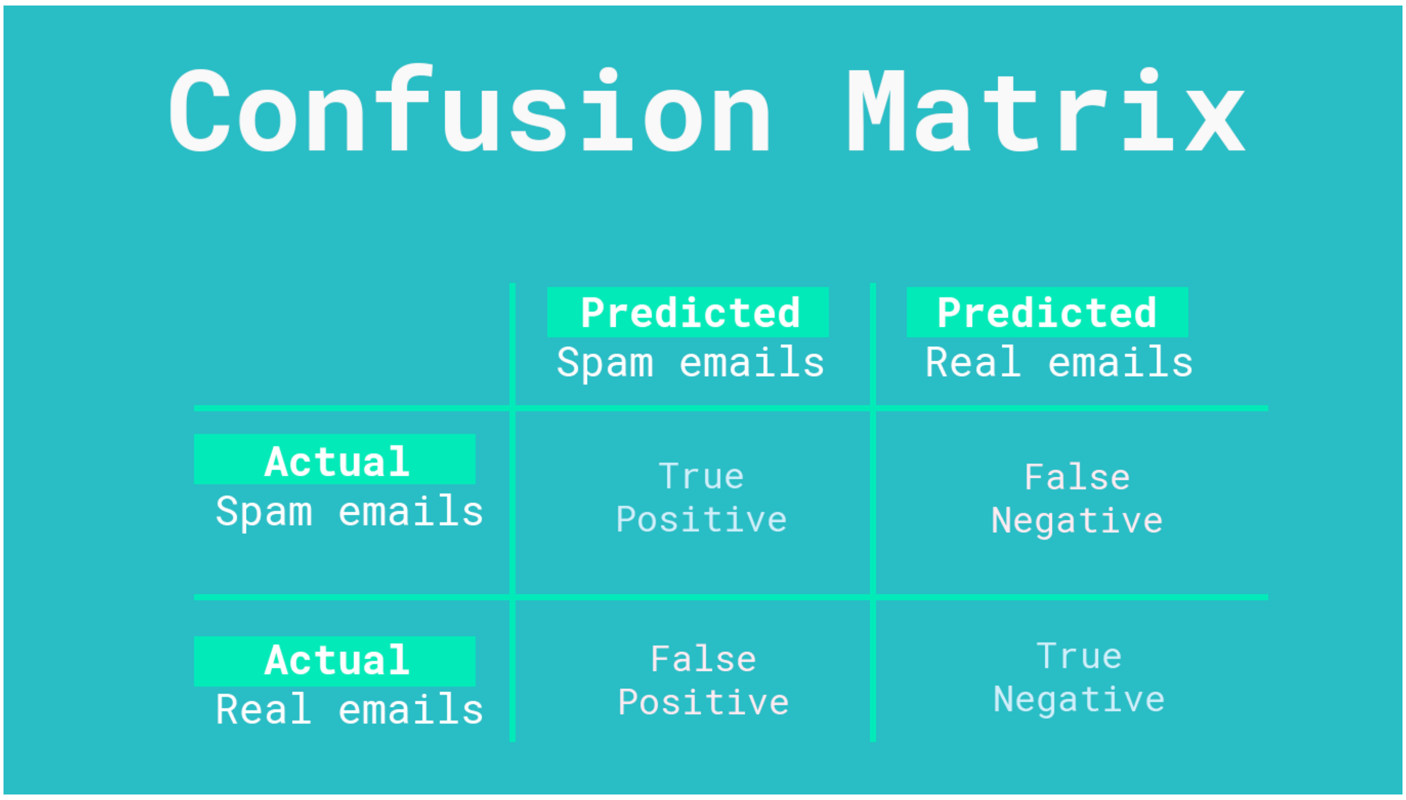
```
[-0.          0.         -0.          0.         -0.          2.95469429
 -0.          0.         -0.         -0.         -0.24795828  0.
 -0.42817442]
```



In [40]:
```python
from sklearn.linear_model import Ridge

x = boston.data
y = boston.target

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=4
2)

ridge = Ridge(alpha=0.1, normalize=True)
ridge.fit(x_train, y_train)
ridge_pred = ridge.predict(x_test)
```

Classification metrics

Confusion Matrix :



Accuracy : $\frac{tp + tn}{tp + tn + fp + fn}$

Precision : $\frac{tp}{tp + fp}$
high precision is meaning not many REAL emails predicted as spam

recall(sensitivity) : $\frac{tp}{tp + fn}$
High recall means predicted most spam emails correctly

F1 Score : $2.\frac{precision.recall}{precision + recall}$

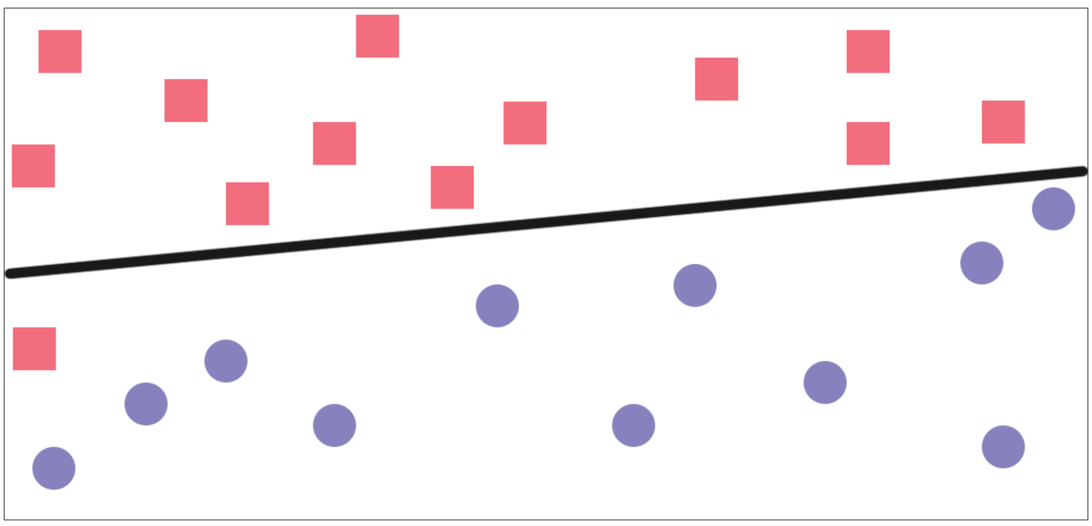
In [50]:
```
from sklearn import datasets

bcd = datasets.load_breast_cancer()

x = bcd.data
y = bcd.target
```

In [51]:
```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=4
2)

knn = KNeighborsClassifier(n_neighbors=8)
knn.fit(x_train, y_train)
y_prediction = knn.predict(x_test)
```

In [43]:
```
print(confusion_matrix(y_test, y_prediction, [0, 1]))
print(classification_report(y_test, y_prediction))
```

```
[[39  4]
 [ 1 70]]
             precision    recall  f1-score   support

          0       0.97      0.91      0.94        43
          1       0.95      0.99      0.97        71

avg / total       0.96      0.96      0.96       114
```

Logistic regression and ROC curve

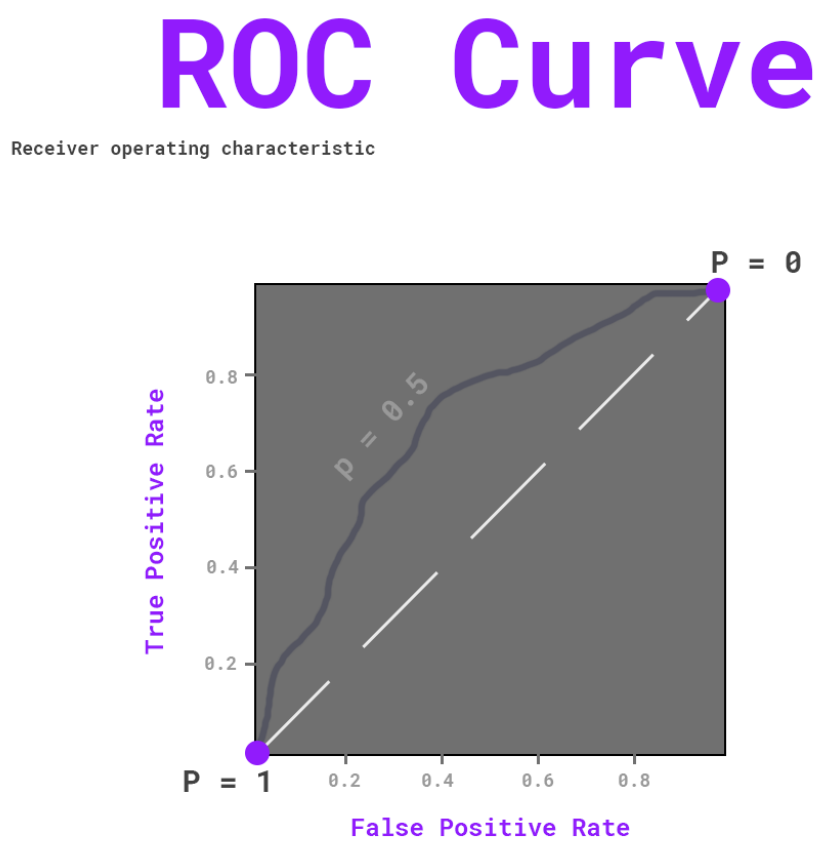


In [44]:
```
from sklearn.linear_model import LogisticRegression

log = LogisticRegression()
log.fit(x_train, y_train)
y_pred = log.predict(x_test)

cm = confusion_matrix(y_test, y_pred)
cm
```

Out[44]:
```
array([[39,  4],
       [ 1, 70]], dtype=int64)
```

In [45]:
```
from sklearn.preprocessing import normalize

cm = normalize(cm,norm='l1',axis=1)

cm_df = pd.DataFrame(cm, columns=bcd.target_names, index=bcd.target_names)
print(cm_df)
```

```
           malignant    benign
malignant   0.906977  0.093023
benign      0.014085  0.985915
```

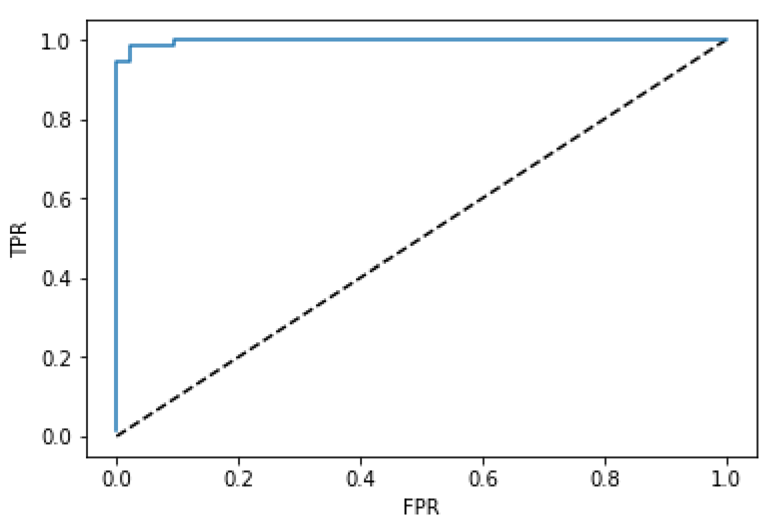


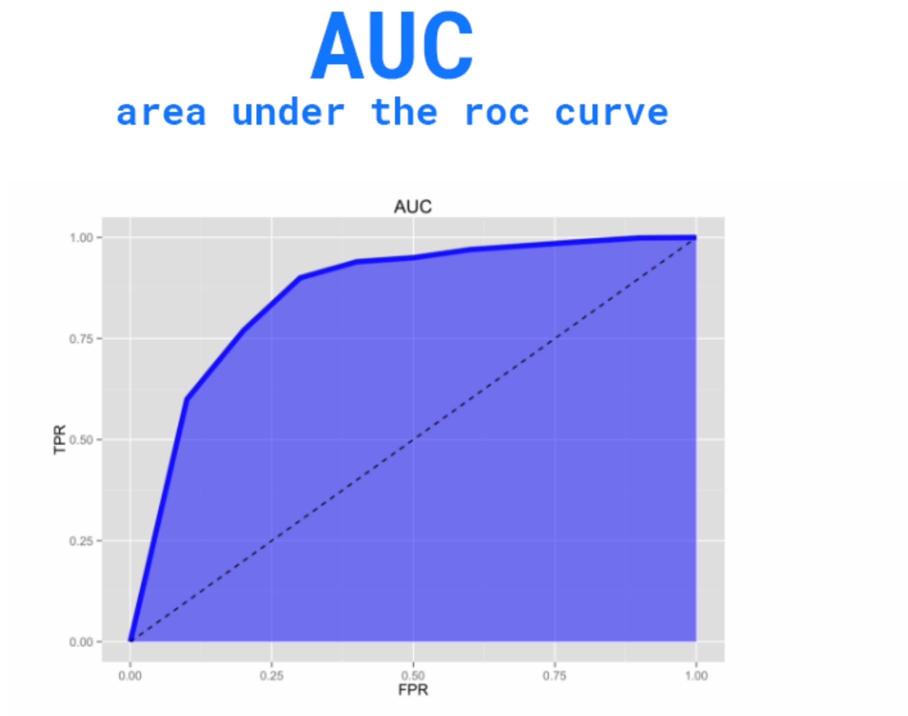
TPR = $\frac{TruePositives}{TruePositives+FalseNegatives}$ = recall(sensitivity)

FPR = $\frac{FalsePositives}{FalsePositives+TrueNegatives}$

ROC.png
images form : http://blog.yhat.com/ (http://blog.yhat.com/)

In [46]:
```
from sklearn.metrics import roc_curve

y_pred_prob = log.predict_proba(x_test)[:,1]
fpr, tpr, threshold = roc_curve(y_test, y_pred_prob)

plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr,tpr)
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.show()
```



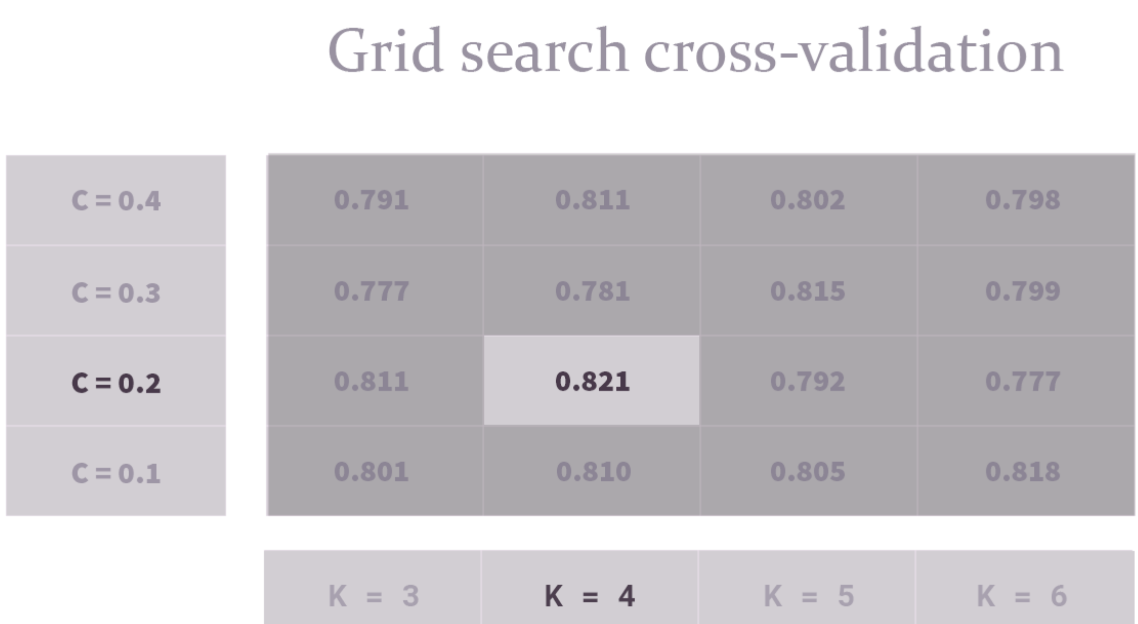


In [48]:
```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_test, y_pred_prob)
```

Out[48]: `0.997707173 27219138`

Hyperparameter Tuning

Grid Search Cross-Validation



In [96]:
```
from sklearn.model_selection import GridSearchCV

param_grid = {'n_neighbors':np.arange(1,50)} # هایپرپارامتر ها را داخل دیکشنری قرار میدهیم


knn = KNeighborsClassifier()
knn_cv = GridSearchCV(knn, param_grid, cv = 5)
knn_cv.fit(x,y)

print(knn_cv.best_params_)
print(knn_cv.best_score_) # Returns the mean accuracy on the given test data and label
s.
```

```
{'n_neighbors': 12}
0.933216168717
```

In [93]:
```
from scipy.stats import randint # randint(1, 9).rvs(2)

#from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import RandomizedSearchCV
#GridSearchCV can be computationally expensive, especially if you are searching over a
 large hyperparameter space and dealing with multiple hyperparameters

param = {"max_depth": [3, None],
              "max_features": randint(1, 9),   # [2, 4, 6, 7]
              "min_samples_leaf": randint(1, 9)}
#Dictionary with parameters names (string) as keys and distributions or lists of parame
ters to try.
#Distributions must provide a rvs method for sampling

tree = DecisionTreeClassifier()
tree_cv = RandomizedSearchCV(tree, param, cv=5) #CV=None, to use the default 3-fold cro
ss validation,

tree_cv.fit(x_train, y_train)

print(tree_cv.best_params_)
print(tree_cv.best_score_)

y_pred = tree_cv.predict(x_test)
score = tree_cv.score(x_test, y_test)

print(score)
```

```
{'max_depth': 3, 'max_features': 8, 'min_samples_leaf': 2}
0.938461538462
0.947368421053
```

Naive Baysian

Naive Bayes is a machine learning method you can use to predict the likelihood that an event will occur given evidence that's present in data .
Bayes' theorem is based on **conditional probability**. The conditional probability helps us calculating the probability that something will happen, given that something else has already happened. Not getting let's understand with few examples.

> The naive Bayes classifier assumes all the features are independent to each other and dont have any correlation .

Example 1

Bayes_41-850x310.png

$$P(Yes \mid Sunny) = \frac{P(Sunny|Yes)*P(Yes)}{P(Sunny)}$$

$P(Sunny \mid Yes) = 3/9 = 0.33$
$P(Sunny) = 5/14 = 0.36$
$P(Yes) = 9/14 = 0.64$

Now

$$P(Yes \mid Sunny) = \frac{0.33*0.64}{0.36} = 0.60$$

which has higher probability.

Naive Bayes uses a similar method to predict the probability of different class based on various attributes

Example 2

https://www.countbayesie.com/blog/2015/2/18/bayes-theorem-with-lego
(https://www.countbayesie.com/blog/2015/2/18/bayes-theorem-with-lego)

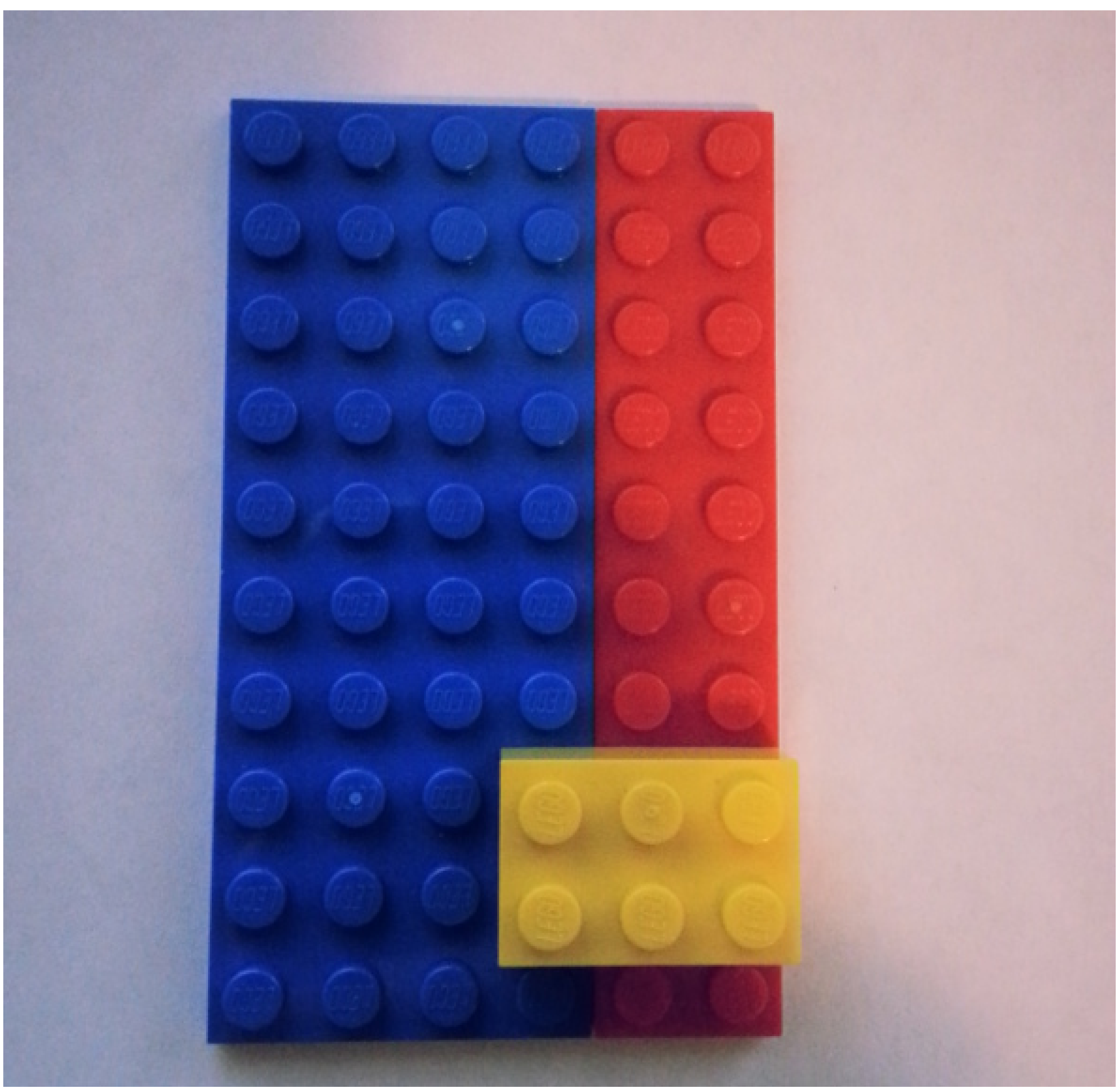


image from : https://www.countbayesie.com/ (https://www.countbayesie.com/)

$$P(\text{blue}) = 40/60 = 2/3$$

$P(\text{red}) = 20/60 = 1/3$

$P(blue) + P(red) = 1$

$P(yellow) = 6/60 = 1/10$

$P(yellow \mid blue)$

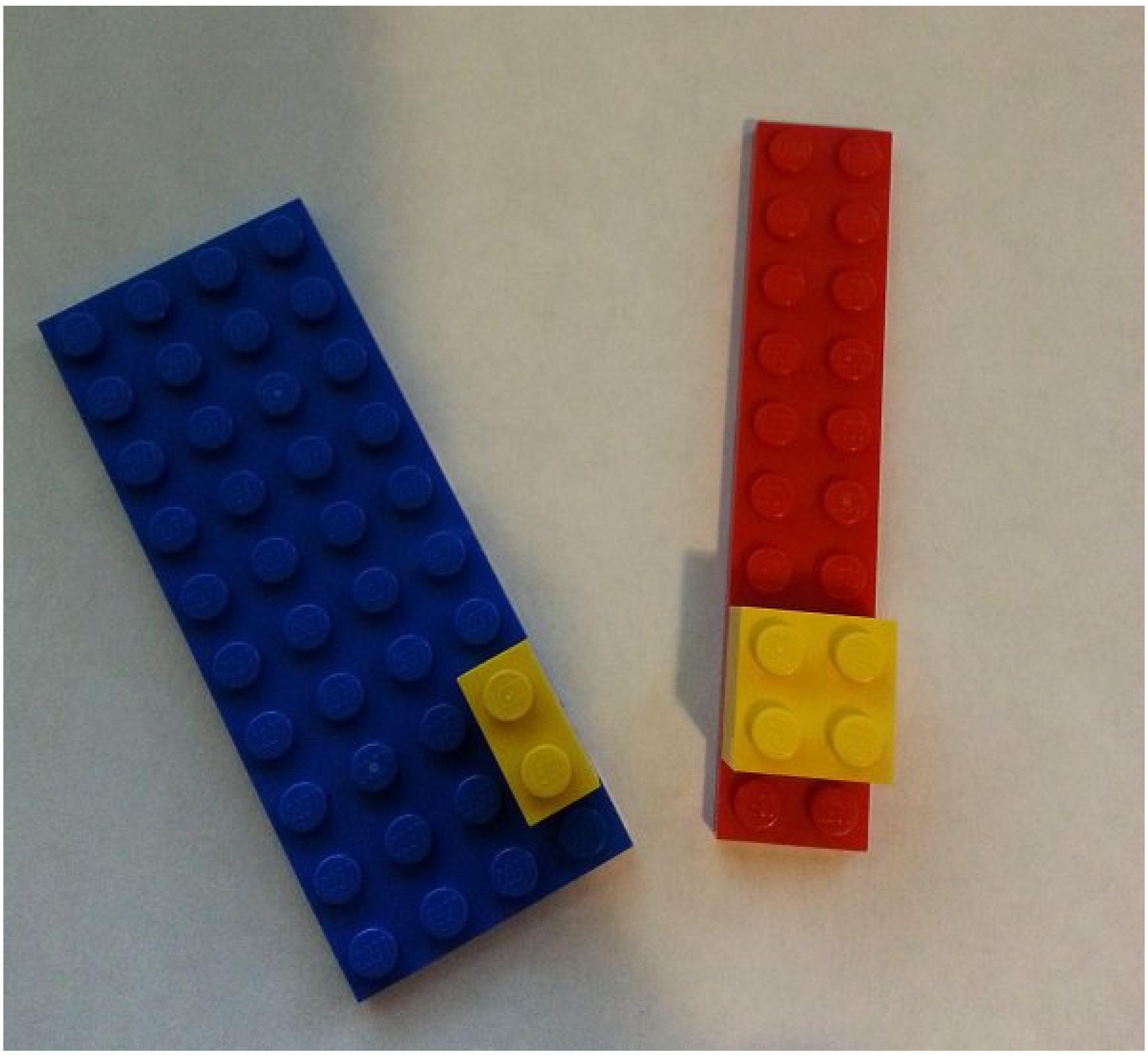
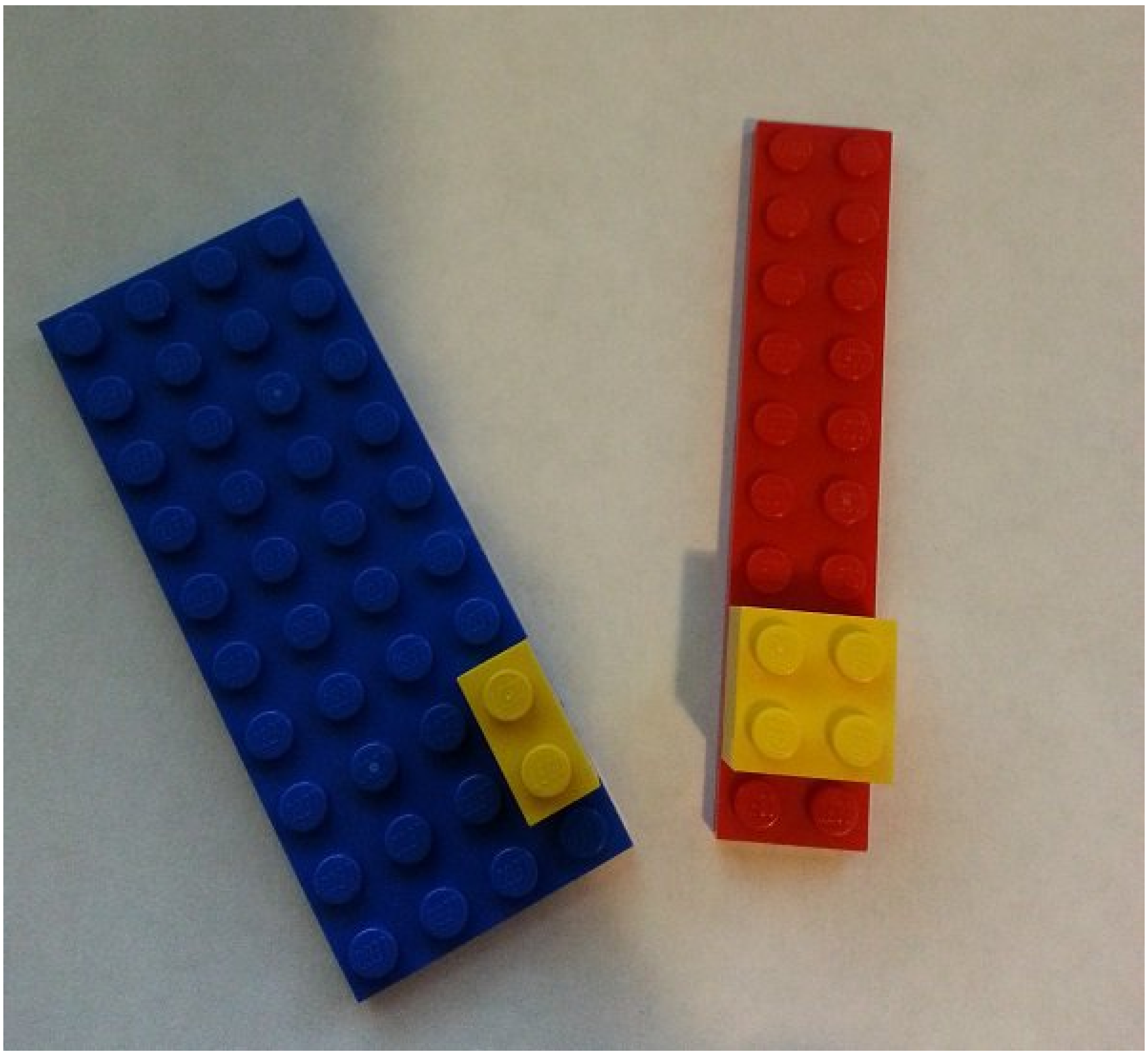


image from : https://www.countbayesie.com/ (https://www.countbayesie.com/)

1- Split the red section off from the blue
2- Get the area of the remaining red space (2 x 10)
3- Get the area of the yellow block on the red space (4)
4- Divide the area of the yellow block by the area of the red block
5- $P(\text{yellow}|\text{red}) = 4/20 = 1/5$



$P(red|yellow)$?

Math approach

$numberOfYellowPegs = P(yellow) \cdot totalPegs = 1/10 \cdot 60 = 6$

$numberOfRedPegs = P(red) \cdot totalPegs = 1/3 \cdot 60 = 20$

$numberOfRedUnderYellow = P(yellow \mid red) \cdot numberOfRedPegs = 1/5 \cdot 20 = 4$

$P(\text{red} \mid \text{yellow}) = \frac{\text{numberOfRedUnderYellow}}{\text{numberOfYellowPegs}} = 4/6 = 2/3$

$P(\text{red} \mid \text{yellow}) = \frac{P(\text{yellow}|\text{red}) \cdot \text{numberOfRedPegs}}{P(\text{yellow}) \cdot \text{totalPegs}}$

$P(\text{red}|\text{yellow}) = \frac{P(\text{yellow}|\text{red}) P(\text{red}) \cdot \text{totalPegs}}{P(\text{yellow}) \cdot \text{totalPegs}}$

$(\text{red}|\text{yellow}) = \frac{P(\text{yellow}|\text{red}) P(\text{red})}{P(\text{yellow})}$

In [36]:
```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
y_pred = gnb.fit(iris.data, iris.target).predict(iris.data)
```