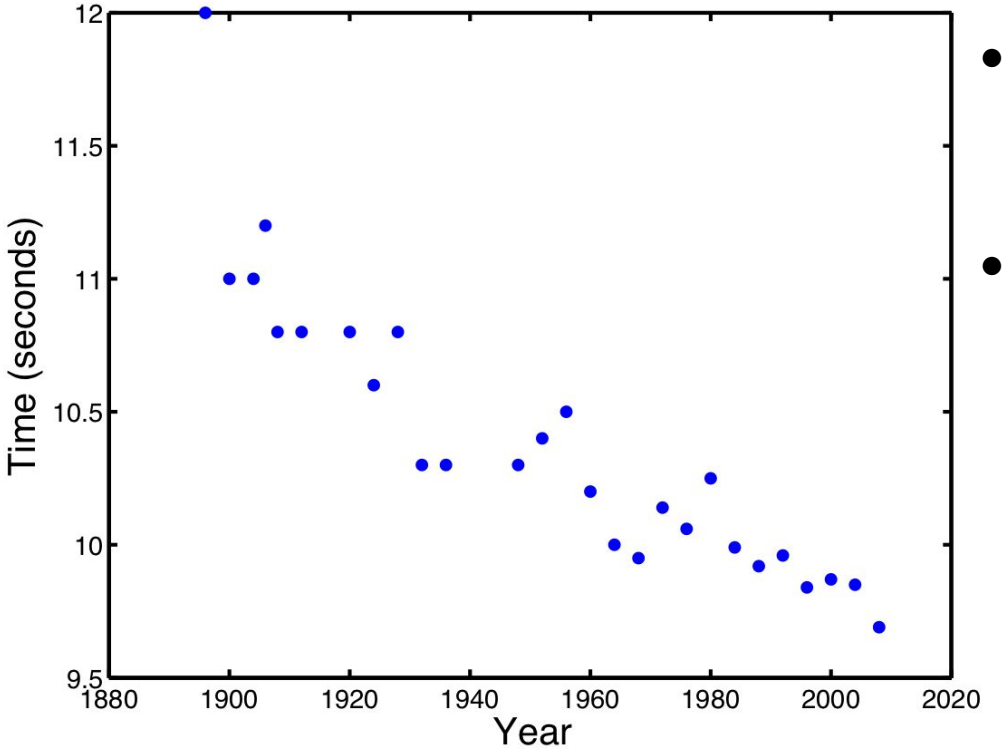
Machine Learning & Artificial Intelligence for Data Scientists: Regression (Part 1)

Ke Yuan
https://kyuanlab.org/
School of Computing Science

Some data and a problem



- Winning times for the men's Olympic 100m sprint, 1896-2008.
- In this lecture, we will use this data to predict the winning time in London 2012

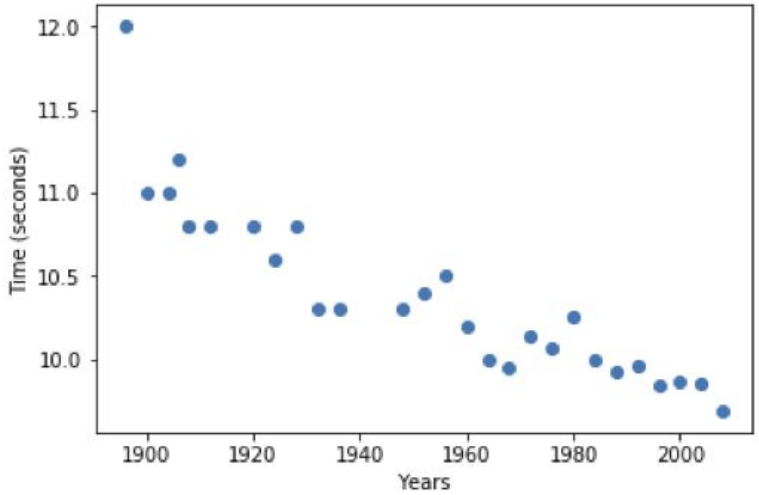
In [13]:
```
import numpy as np
%matplotlib inline
import pylab as plt

data = np.loadtxt('olympic100m.txt', delimiter=',') # load olympic data
data
```

Out[13]:
```
array([[1896.  ,   12.  ],
       [1900.  ,   11.  ],
       [1904.  ,   11.  ],
       [1906.  ,   11.2 ],
       [1908.  ,   10.8 ],
       [1912.  ,   10.8 ],
       [1920.  ,   10.8 ],
       [1924.  ,   10.6 ],
       [1928.  ,   10.8 ],

       [1932.  ,   10.3 ],
       [1936.  ,   10.3 ],
       [1948.  ,   10.3 ],
       [1952.  ,   10.4 ],
       [1956.  ,   10.5 ],
       [1960.  ,   10.2 ],
       [1964.  ,   10.  ],
       [1968.  ,    9.95],
       [1972.  ,   10.14],
       [1976.  ,   10.06],
       [1980.  ,   10.25],
       [1984.  ,    9.99],
       [1988.  ,    9.92],
       [1992.  ,    9.96],
       [1996.  ,    9.84],
       [2000.  ,    9.87],
       [2004.  ,    9.85],
       [2008.  ,    9.69]])
```

Let's look at the data

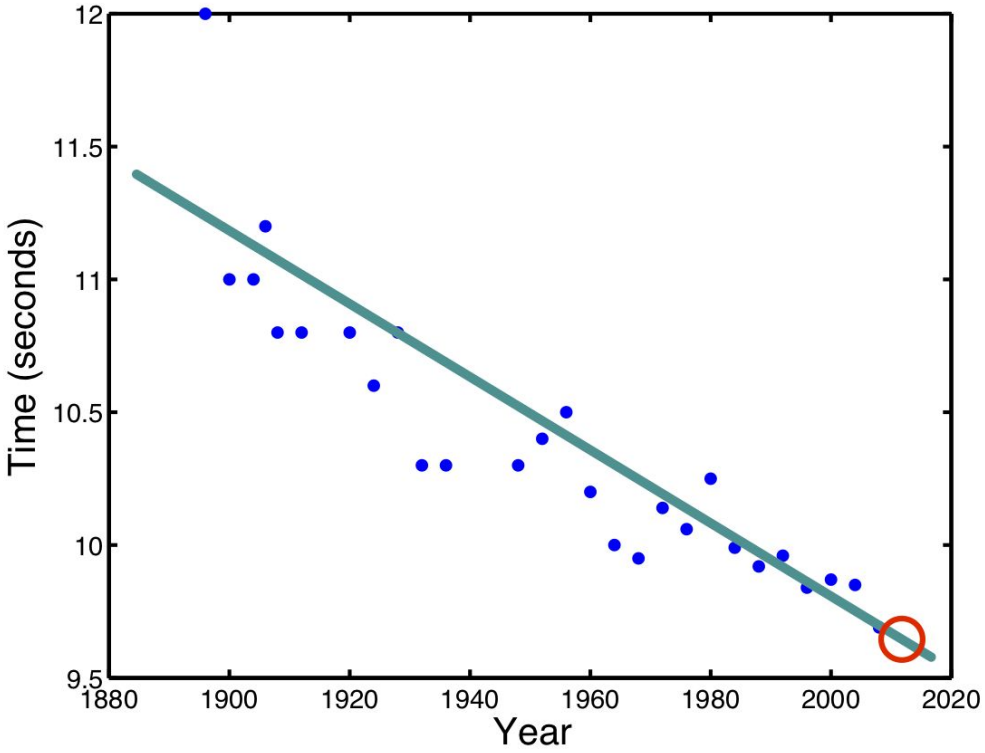
In [15]:
```
x =  data[:,0] # name years as x
t =  data[:,1] # name time as t

plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

Out[15]:
```
Text(0, 0.5, 'Time (seconds)')
```



Our first scatter plot

Draw a line through it!



Overview: Simple Linear Regression

- Introduce the idea of building models.
- Talk about assumptions.
- Use a linear model.
- What constitutes a good model?
- Find the best linear model.
- Use it to predict the winning time in 2012.

Assumptions

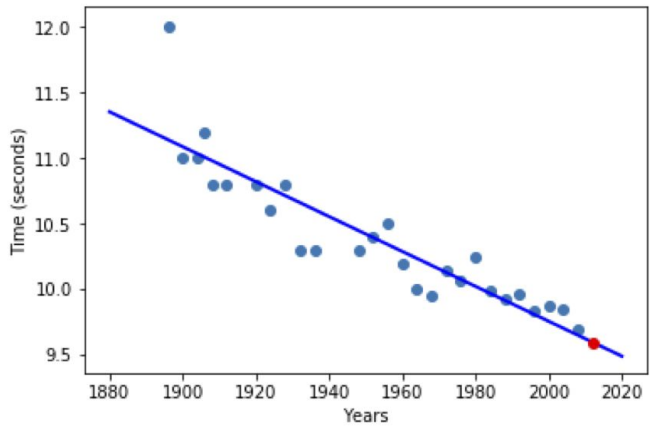
---

1. That there exists a relationship between Olympic year and winning time.
2. That this relationship is linear (i.e. a straight line).
3. That this relationship will continue into the future.

In [24]:
```
# fit a straight line
w_0 = 36.4164559025
w_1 = -0.013330885711

x_test = np.linspace(1880,2020, 100) # generate new x to plot the fitted line. Not
e better not to use the original x !
f_test = w_0 + w_1 * x_test
plt.plot(x_test,f_test,'b-',linewidth=2) # plot the fitted data
plt.plot(2012, w_0 + w_1 * 2012, 'ro')

plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

Out[24]:
```
Text(0, 0.5, 'Time (seconds)')
```

Draw a line through it!



Let's reflect on the task

Attributes and targets

Typically in Supervised Machine Learning, we have a set of attributes and corresponding targets:

- **Attributes:** Olympic year.
- **Targets:** Winning time.

Key definitions

Variables

Mathematically, each is described by a variable:

- Olympic year: x.
- Winning time: t.

Key definitions

Model

Our goal is to create a model.

- This is a function that can relate x to t.

$$t = f(x)$$

- Hence, we can work out t when $x = 2012$.

Key definitions

Data

We're going to create the model from data:

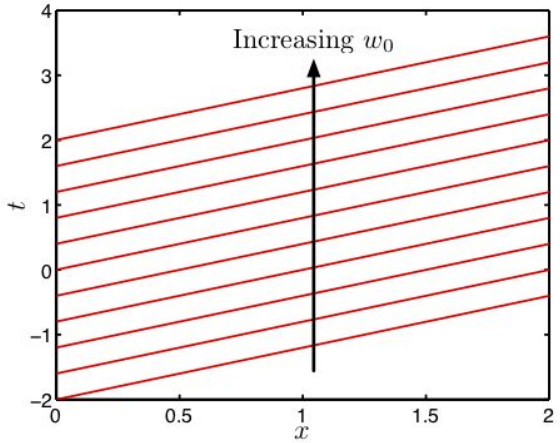
- N attribute-response pairs, (x_n, t_n)
- e.g. $(1896, 12s), (1900, 11s), \ldots, (2008, 9.69s)$
- $x_1 = 1896,\ t_1 = 12$, etc

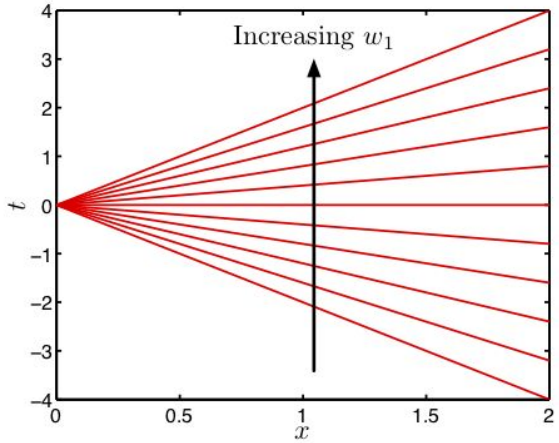
Often called **training** data

What is a model?

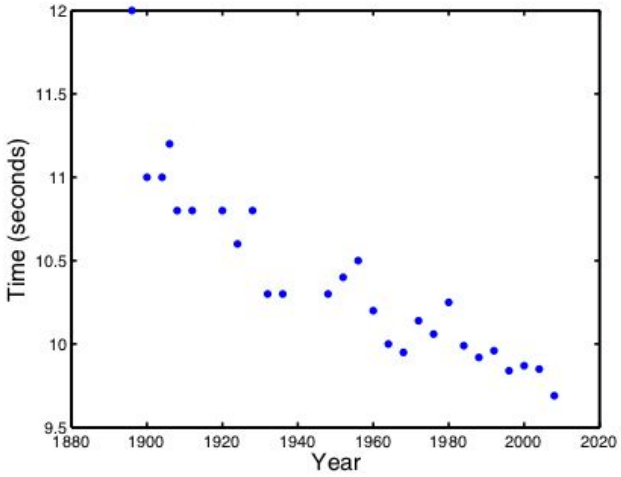
$$t = f(x) = w_0 + w_1 x = f(x; w_0, w_1)$$

- w_0 and w_1 are *parameters* of the model.
- They determine the properties of the line.

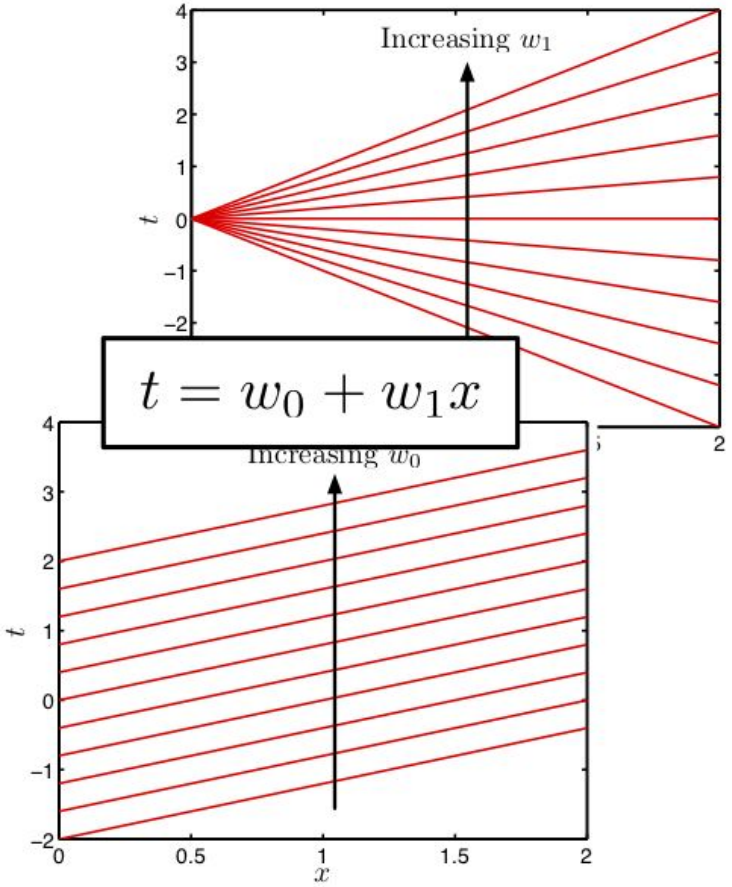




We have data and a family of models:



?

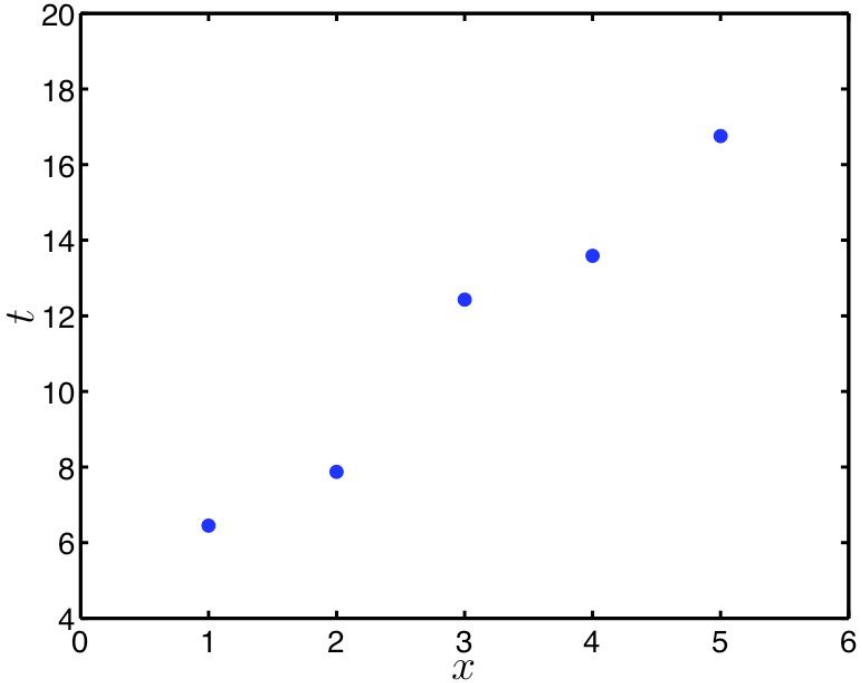


What is Learning?

How good is a particular w_0, w_1?

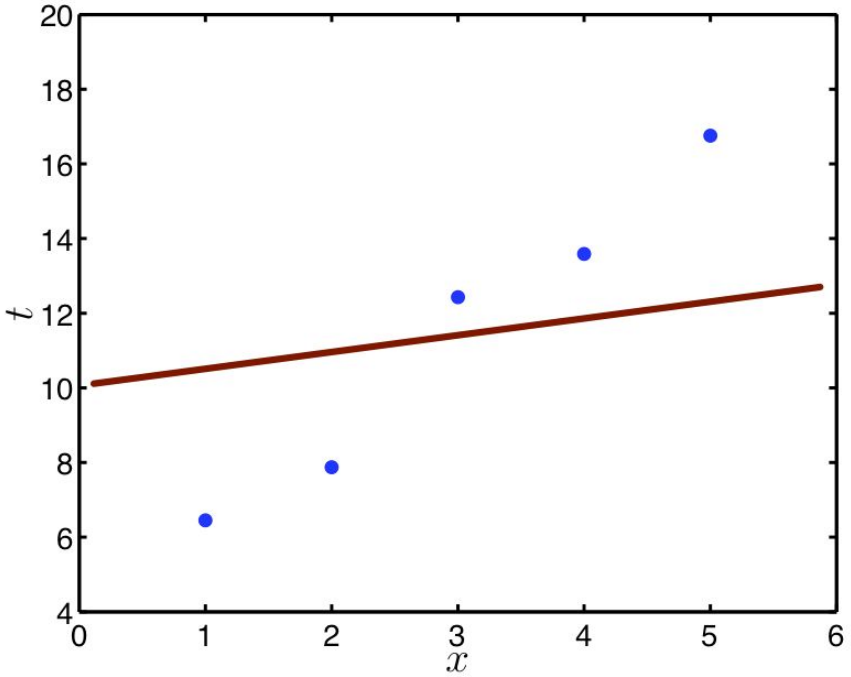
- How good is a particular line (w_0, w_1)?
- We need to be able to provide a numerical value of goodness for any w_0, w_1.
  - How good is $w_0 = 5, w_1 = 0.1$?
  - Is $w_0 = 5, w_1 = -0.1$ better or worse?
- Once we can answer these questions, we can search for the best w_0, w_1 pair.

Loss



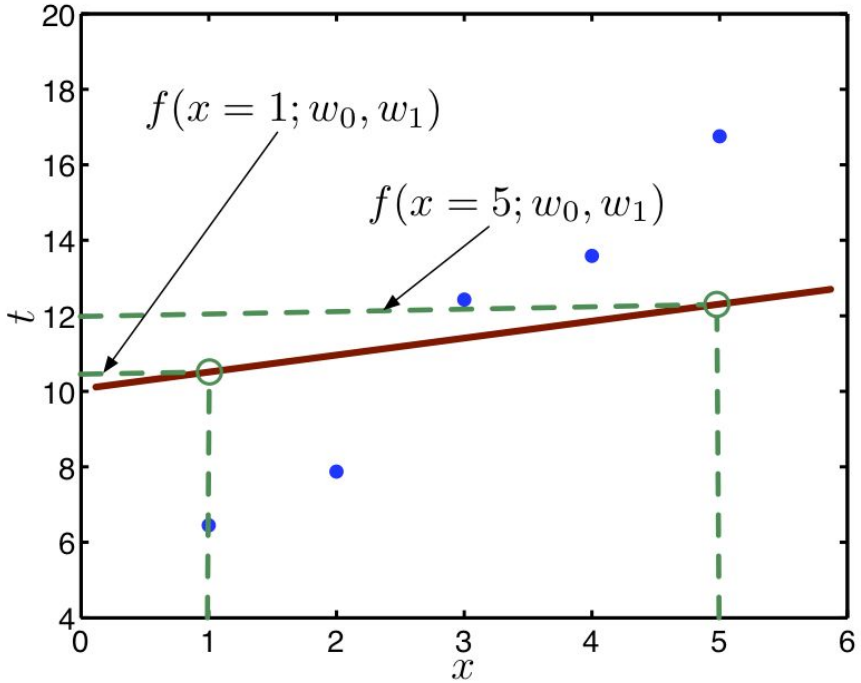
Some different data.

Loss



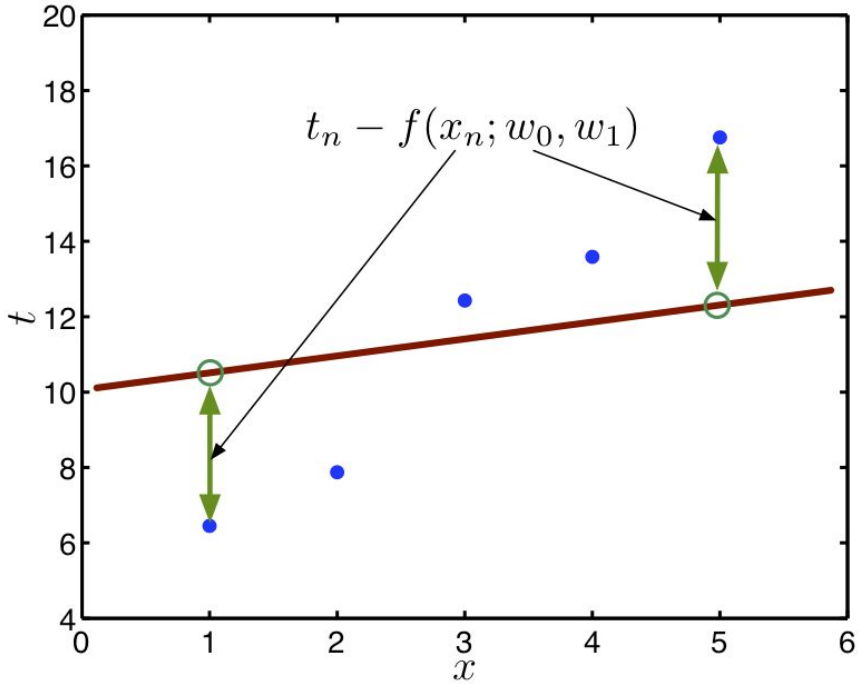
Given w_0 and w_1 you can draw a line.

Loss



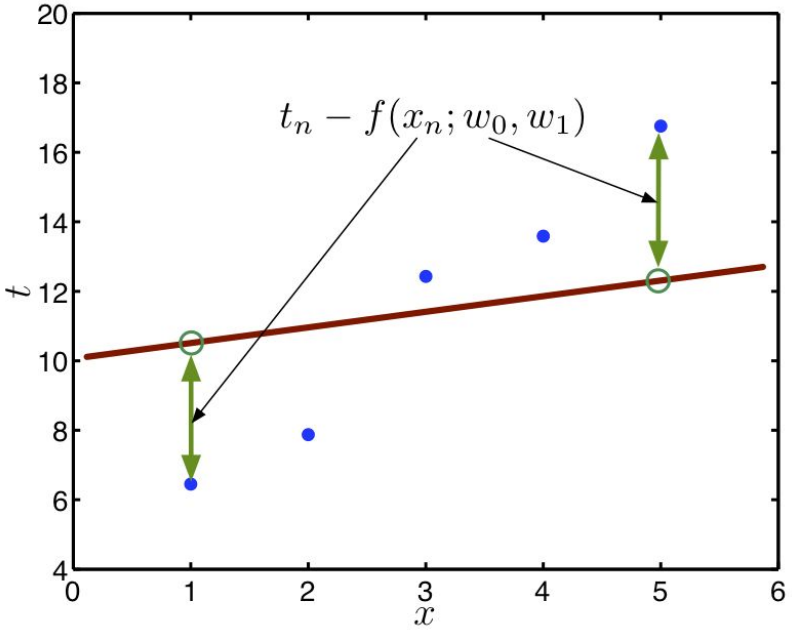
This means that we can compute $f(x_n; w_0, w_1)$ for each x_n.

Loss



$f(x_n; w_0, w_1)$ can be compared with the truth, t_n.

Squared Loss



$f(x_n; w_0, w_1)$ can be compared with the truth, t_n.
$(t_n - f(x_n; w_0, w_1))^2$ tells us how *badly* we model (x_n, t_n).

Squared Loss

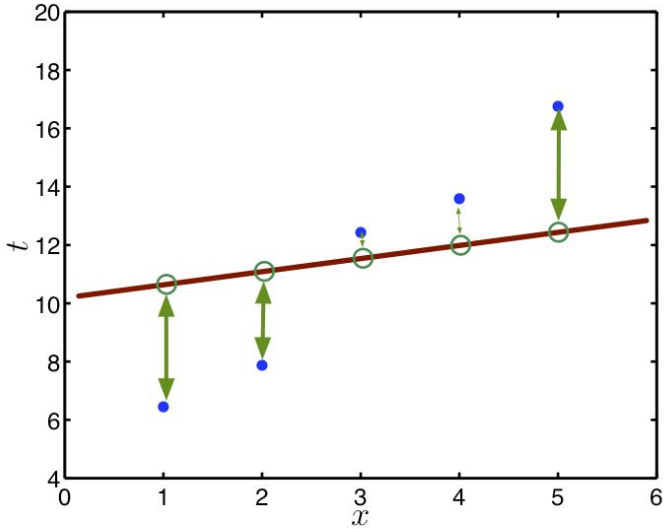
–

- The *Squared loss* of training point n is defined as:

$$\mathcal{L}_n = (t_n - f(x_n; w_0; w_1))^2$$

- It is the squared difference between the true response (winning time), t_n when the input is x_n and the response predicted by the model, $f(x_n; w_0, w_1) = w_0 + w_1 x_n$.
- The lower $\mathcal{L}_n$, the closer the line at x_n passes to t_n.

Averaged Squared Loss

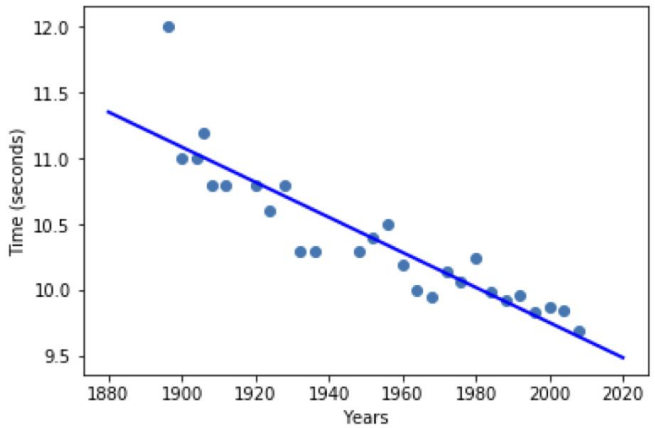


Average the loss at each training point to give single figure for all data:

$$\mathcal{L} = \frac{1}{N} \sum_{n=1}^{N} (t_n - f(x_n; w_0, w_1))^2$$

In [4]:
```
# fit a straight line
w_0 = 36.4164559025
w_1 = -0.013330885711

x_test = np.linspace(1880,2020, 100) # generate new x to plot the fitted line. Not
e better not to use the original x !
f_test = w_0 + w_1 * x_test
plt.plot(x_test,f_test,'b-',linewidth=2) # plot the fitted data

plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

Out[4]:
```
Text(0, 0.5, 'Time (seconds)')
```



Compare two different models

Model 1 :
w_0 = 36.4164559025
w_1 = -0.013330885711

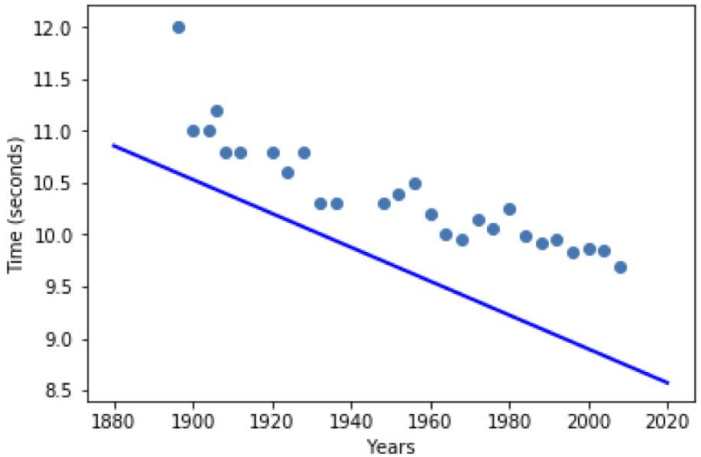
In [5]:
```
sum((t-36.4164559025 - (-0.013330885711)*x)**2)/t.shape[0]
```

Out[5]:
```
0.05030711047565771
```

In [6]:
```
# fit a straight line
w_0 = 41.5
w_1 = -0.0163

x_test = np.linspace(1880,2020, 100) # generate new x to plot the fitted line. Not
e better not to use the original x !
f_test = w_0 + w_1 * x_test
plt.plot(x_test,f_test,'b-',linewidth=2) # plot the fitted data

plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

Out[6]:
```
Text(0, 0.5, 'Time (seconds)')
```

Compare two different models



Model 2 :
w_0 = 41.5
w_1 = -0.0163

In [7]:
```
sum((t-41.5- (- 0.013330885711)*x)**2)/t.shape[0]
```

Out[7]:
```
25.892727700891623
```

Model fitting

In [52]:
```
from sklearn.linear_model import LinearRegression # import

x = x[:,None] # 27 x 1 array
t = t[:,None] # 27 x 1 array

reg = LinearRegression().fit(x, t)
```

In [53]:
```
[reg.intercept_, reg.coef_]
```

Out[53]:
```
[array([36.4164559]), array([[-0.01333089]])]
```

In [54]:
```
reg.predict(np.array([[2012]]))
```

Out[54]:
```
array([[9.59471385]])
```

Summary

- Introduced some ideas about modelling.
- Found some data.
- Derived a way of saying how good a model is.
- Found an expression for the best model.
- Used this to fit a model to the Olympic data.
- Made a prediction for the winning time in 2012.

Assumptions again

---

1. That there exists a relationship between Olympic year and winning time.
2. That this relationship is linear (i.e. a straight line).
3. That this relationship will continue into the future.

Assumptions are wrong

- Relationship is clearly not perfectly linear.
- Winning time cannot decrease forever - it must be positive.
- It can't increase forever into the past.

The model is 'wrong' but it might still be useful. How useful depends on the questions we wish to answer.

Machine Learning & Artificial Intelligence for Data Scientists: Regression (Part 2)

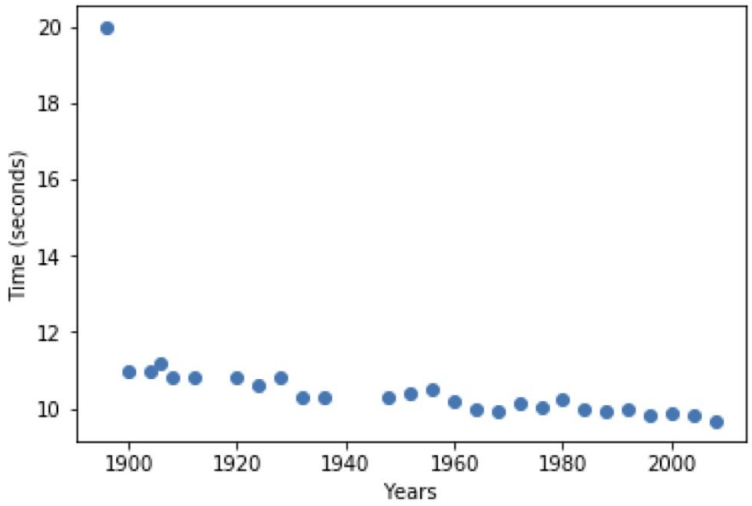
Ke Yuan
https://kyuanlab.org/
School of Computing Science

1-2
Recap

- Introduced some ideas about modelling.
- Found some data.
- Derived a way of saying how good a model is.
- Found an expression for the best model.
- Used this to fit a model to the Olympic data.
- Made a prediction for the winning time in 2012.

In [69]:
```
outlier_idx = np.array([0])
t_outlier = t*1
t_outlier[outlier_idx] = 20

plt.scatter(x,t_outlier) # draw a scatter plot
plt.xlabel('Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

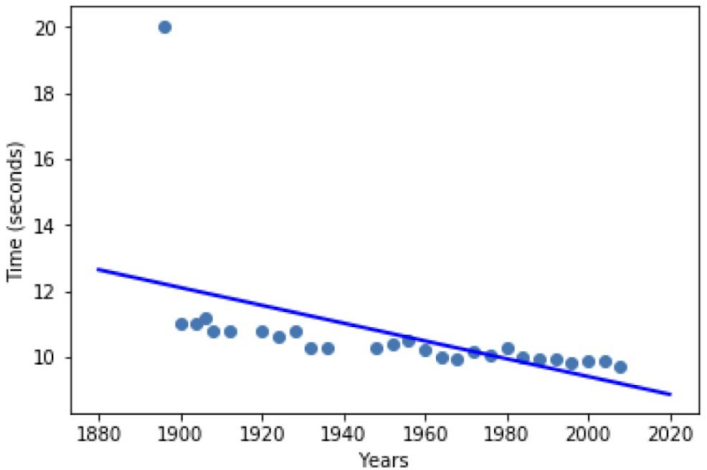
Out[69]:
```
Text(0, 0.5, 'Time (seconds)')
```



Let's add some outliers

In [81]:
```
from sklearn.linear_model import LinearRegression # import

reg = LinearRegression().fit(x, t_outlier)

x_test = np.linspace(1880,2020, 100)[:,None] # test data
f_test = reg.predict(x_test)
plt.plot(x_test,f_test,'b-',linewidth=2) # plot the fitted data

plt.scatter(x,t_outlier) # draw a scatter plot
plt.xlabel('Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

Out[81]:
```
Text(0, 0.5, 'Time (seconds)')
```



Outliers hurt simple linear regression badly

In [16]:
```
[reg.intercept_, reg.coef_]
```

Out[16]:
```
[array([63.32175978]), array([[-0.02695996]])]
```

Going beyond straight line: Polynomial Regression

$$t = w_0 + w_1 x + w_2 x^2 + w_3 x^2 + \ldots + w_K x^K = \sum_{k=0}^{K} w_k x^k$$

- To find $\widehat{w_0}, \ldots, \widehat{w_K}$:
  - Define loss $\mathcal{L} = \frac{1}{N} \sum_{n=1}^{N} \left(t_n - \sum_{k=0}^{K} w_k x^k \right)^2$
  - Differentiate loss with respect to every parameter
  - Set to zero and solve (K simultaneous equations)
- Very tedious! Use vector/matrix notation instead.

Vector/Matrix form: This is still Linear Regression!

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_K \end{bmatrix}, \mathbf{x}_n = \begin{bmatrix} 1 \\ x_n \\ x_n^2 \\ \vdots \\ x_n^K \end{bmatrix}, \mathbf{X} = \begin{bmatrix} 1 & x_1^1 & x_1^2 & \dots & x_1^K \\ 1 & x_2^1 & x_2^2 & \dots & x_2^K \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N^1 & x_N^2 & \dots & x_N^K \end{bmatrix}$$

$$t = \mathbf{w}^\top \mathbf{x}, \quad \mathcal{L} = \frac{1}{N}(\mathbf{t} - \mathbf{X}\mathbf{w})^\top (\mathbf{t} - \mathbf{X}\mathbf{w})$$

Least Square Solution

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & 1896 \\ 1 & 1900 \\ \vdots & \\ 1 & 2008 \end{bmatrix}, \quad \mathbf{t} = \begin{bmatrix} 12.00 \\ 11.00 \\ \vdots \\ 9.85 \end{bmatrix}$$

$$\widehat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t} = \begin{bmatrix} 36.416 \\ -0.0133 \end{bmatrix}$$

Construct polynomial matrix

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^K \\ 1 & x_2 & x_2 & \dots & x_2^K \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^K \end{bmatrix}$$

In [7]:
```
def make_polynomial (x, maxorder): # The np.hstack function can be very helpful
    X = np.ones_like(x)
    for i in range(1,maxorder+1):
        X = np.hstack((X,x**i))
    return(X)
```

In [8]:
```
poly_order = 3
poly_X = make_polynomial(x, poly_order)
poly_X
```

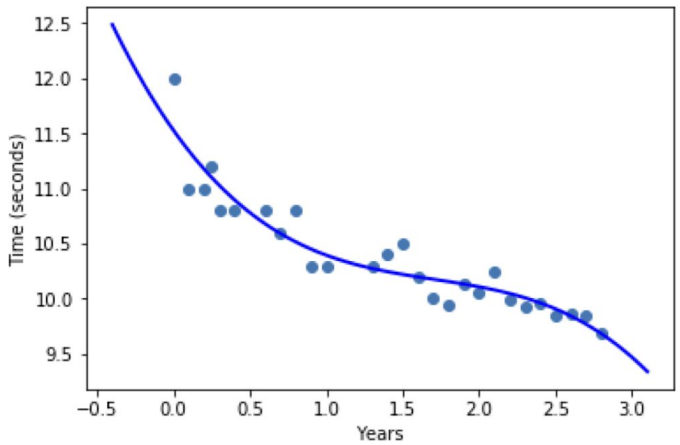
Out[8]:
```
array([[1.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00],
       [1.0000e+00, 1.0000e-01, 1.0000e-02, 1.0000e-03],
       [1.0000e+00, 2.0000e-01, 4.0000e-02, 8.0000e-03],
       [1.0000e+00, 2.5000e-01, 6.2500e-02, 1.5625e-02],
       [1.0000e+00, 3.0000e-01, 9.0000e-02, 2.7000e-02],
       [1.0000e+00, 4.0000e-01, 1.6000e-01, 6.4000e-02],
       [1.0000e+00, 6.0000e-01, 3.6000e-01, 2.1600e-01],
       [1.0000e+00, 7.0000e-01, 4.9000e-01, 3.4300e-01],
       [1.0000e+00, 8.0000e-01, 6.4000e-01, 5.1200e-01],
       [1.0000e+00, 9.0000e-01, 8.1000e-01, 7.2900e-01],
       [1.0000e+00, 1.0000e+00, 1.0000e+00, 1.0000e+00],
       [1.0000e+00, 1.3000e+00, 1.6900e+00, 2.1970e+00],
       [1.0000e+00, 1.4000e+00, 1.9600e+00, 2.7440e+00],
       [1.0000e+00, 1.5000e+00, 2.2500e+00, 3.3750e+00],
       [1.0000e+00, 1.6000e+00, 2.5600e+00, 4.0960e+00],
       [1.0000e+00, 1.7000e+00, 2.8900e+00, 4.9130e+00],
       [1.0000e+00, 1.8000e+00, 3.2400e+00, 5.8320e+00],
       [1.0000e+00, 1.9000e+00, 3.6100e+00, 6.8590e+00],
       [1.0000e+00, 2.0000e+00, 4.0000e+00, 8.0000e+00],
       [1.0000e+00, 2.1000e+00, 4.4100e+00, 9.2610e+00],
       [1.0000e+00, 2.2000e+00, 4.8400e+00, 1.0648e+01],
       [1.0000e+00, 2.3000e+00, 5.2900e+00, 1.2167e+01],
       [1.0000e+00, 2.4000e+00, 5.7600e+00, 1.3824e+01],
       [1.0000e+00, 2.5000e+00, 6.2500e+00, 1.5625e+01],
       [1.0000e+00, 2.6000e+00, 6.7600e+00, 1.7576e+01],
       [1.0000e+00, 2.7000e+00, 7.2900e+00, 1.9683e+01],
       [1.0000e+00, 2.8000e+00, 7.8400e+00, 2.1952e+01]])
```

Construct polynomial matrix

In [41]:
```
poly_order = 3
X_train = make_polynomial(x, poly_order)
poly_reg = LinearRegression().fit(X_train, t) # Fit a linear model
print('loss at order 3:', np.mean((t-poly_reg.predict(X_train))**2 ) )
X_test = make_polynomial(x_test, poly_order) # construct the polynomial matrix for
test data
f_test = poly_reg.predict(X_test)
plt.plot(x_test,f_test,'b-',linewidth=2) # plot the fitted data
plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

```
loss at order 3: 0.02961132122019676
```

Out[41]:
```
Text(0, 0.5, 'Time (seconds)')
```

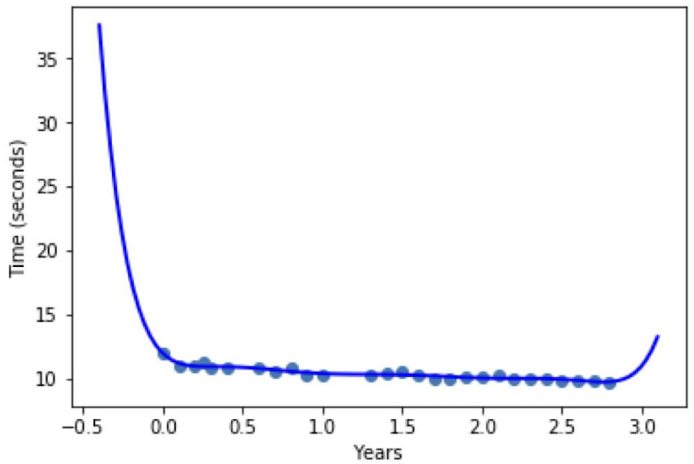


Fit the model using the same formula

In [42]:
```
poly_order = 8
X_train = make_polynomial(x, poly_order)
poly_reg = LinearRegression().fit(X_train, t)
print('loss at order 8:', np.mean((t-poly_reg.predict(X_train))**2 ) )
X_test = make_polynomial(x_test, poly_order)
f_test = poly_reg.predict(X_test)
plt.plot(x_test,f_test,'b-',linewidth=2) # plot the fitted data
plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

```
loss at order 8: 0.016981387841969484
```

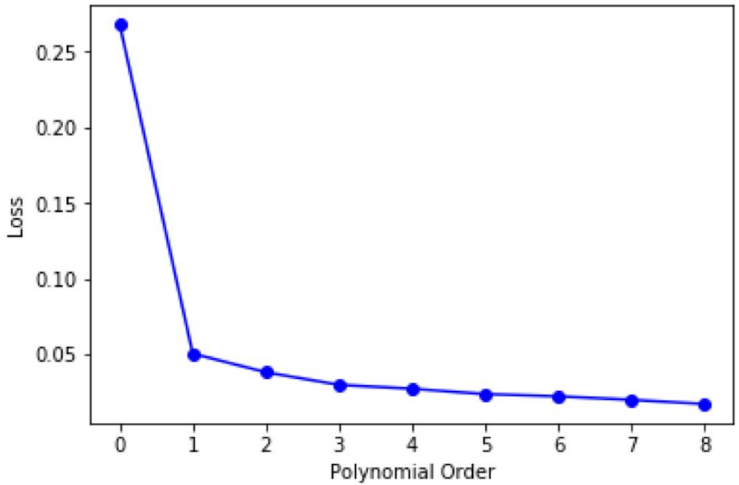
Out[42]:
```
Text(0, 0.5, 'Time (seconds)')
```



What about higher order?

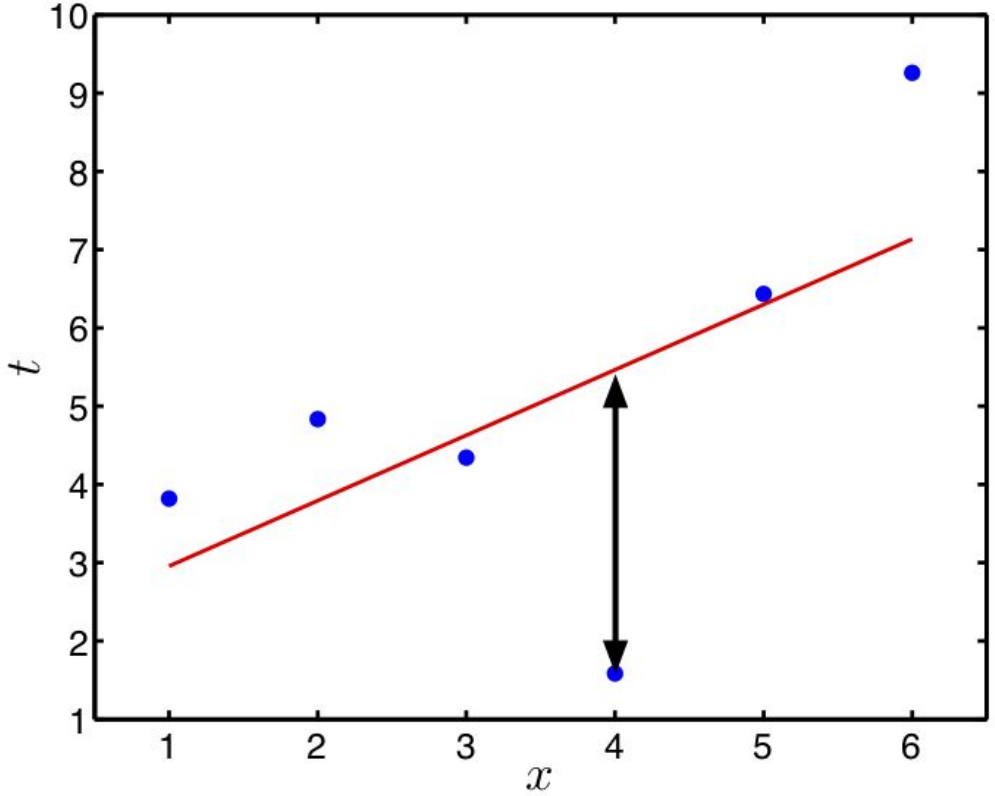
In [84]:
```
all_loss = np.zeros(9)
for i in range(9):
    poly_order = i
    X_train = make_polynomial(x, poly_order)
    poly_reg = LinearRegression().fit(X_train, t)
    all_loss[i] = np.mean((t-poly_reg.predict(X_train))**2 )

plt.plot(all_loss, 'bo-')
plt.xlabel('Polynomial Order') # always label x&y-axis
plt.ylabel('Loss') # always label x&y-axis
```

Out[84]:
```
Text(0, 0.5, 'Loss')
```



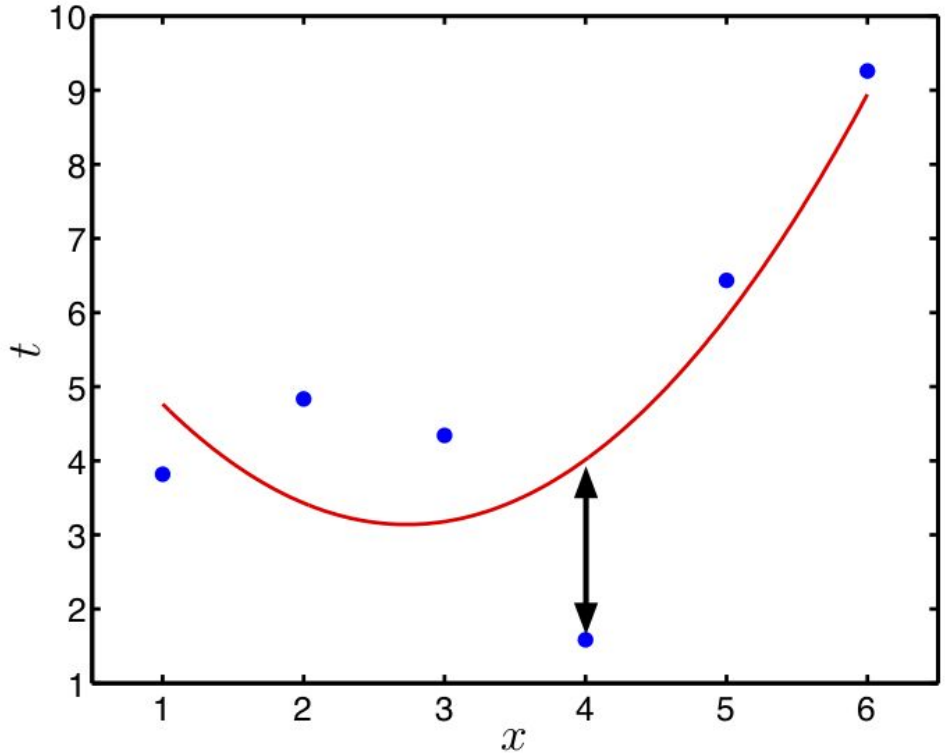
Loss always decreases as the model is made more complex

Data comes from $t = x$ with some *noise* added:



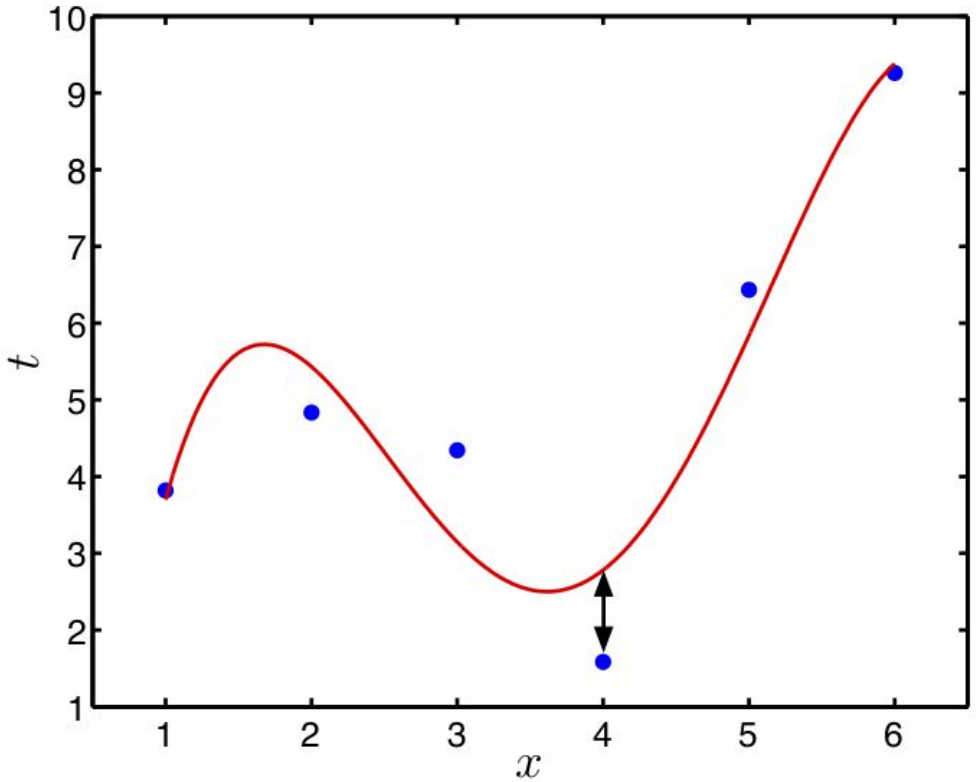
Linear model $t = w_0 + w_1 x$.

Data comes from $t = x$ with some *noise* added:



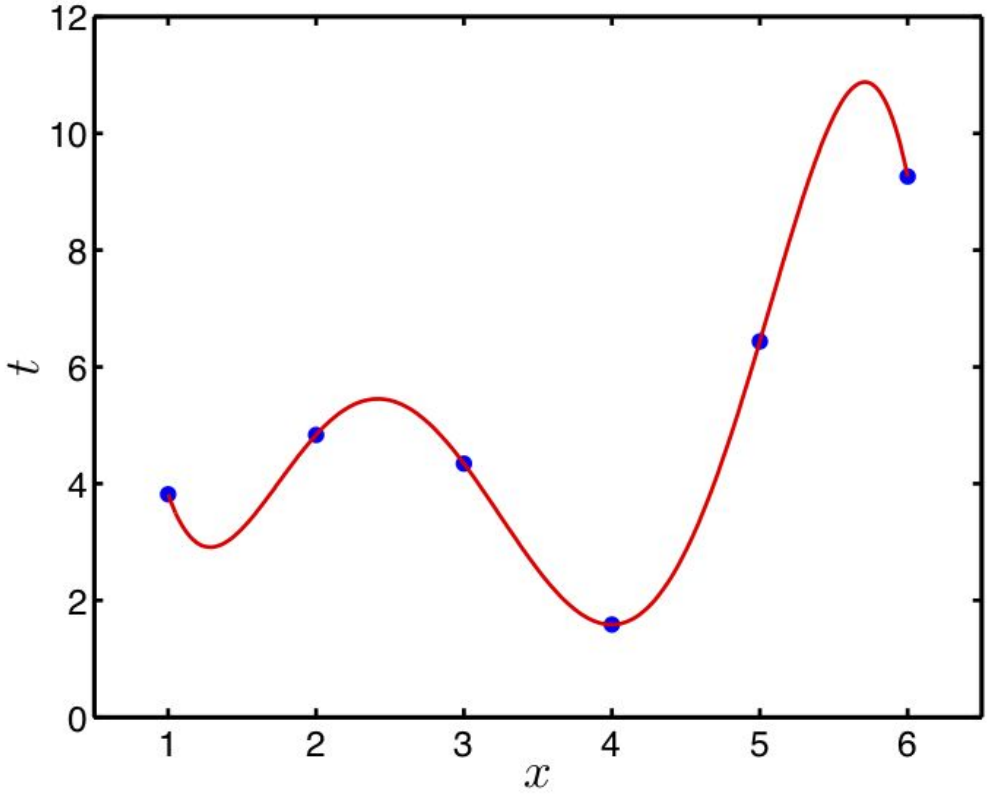
Quadratic model $t = w_0 + w_1 x + w_2 x^2$.

Data comes from $t = x$ with some *noise* added:



Fourth order $t = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$.

Data comes from $t = x$ with some *noise* added:

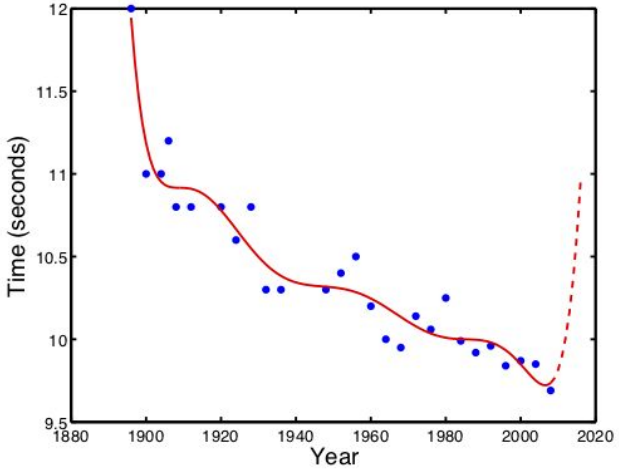


Fifth order $t = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + w_5 x^5$.

Generalisation and over-fitting

There is a trade-off between generalisation (predictive ability) and over-fitting (decreasing the loss).

- Fitting a model perfectly to the training data is likely to lead to poor predictions because there will almost always be *noise* present.



Noise

Not necessarily 'noise', just things we can't, or don't need to model.

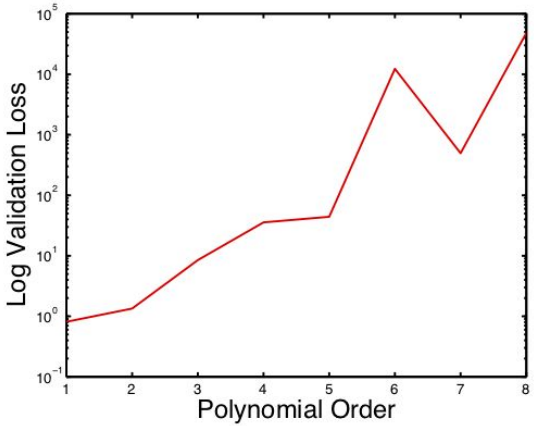
How do we choose the right model complexity? Where can we get more data?

- We have N input-response pairs for training:

$$(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N).$$

- We could use $N - C$ pairs to find $\widehat{\mathbf{w}}$ for several models.
- Choose the model that makes best predictions on remaining C pairs.
  - The $N - C$ pairs constitute *training data*.
  - The C pairs are known as *validation data*.
- Example – use Olympics pre 1980 to train and post 1980 to validate.

Validation example



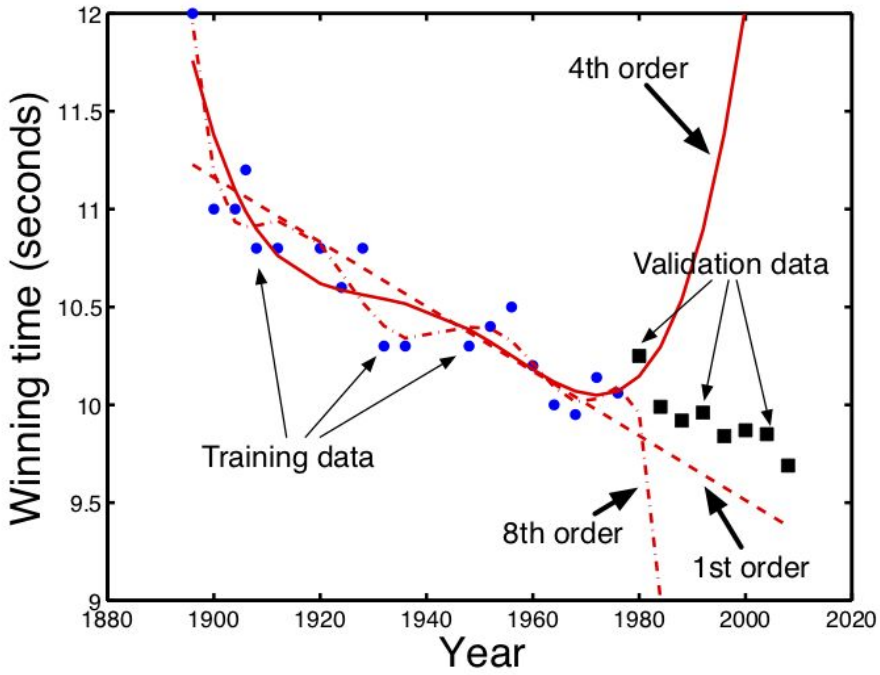
Predictions evaluated using validation loss:

$$\mathcal{L}_v = \frac{1}{C}\sum_{c=1}^{C}(t_c - \mathbf{w}^{\mathsf{T}}\mathbf{x}_c)^2$$

Best model?

Results suggest that a first order (linear) model ($t = w_0 + w_1 x$) is best.

Validation example

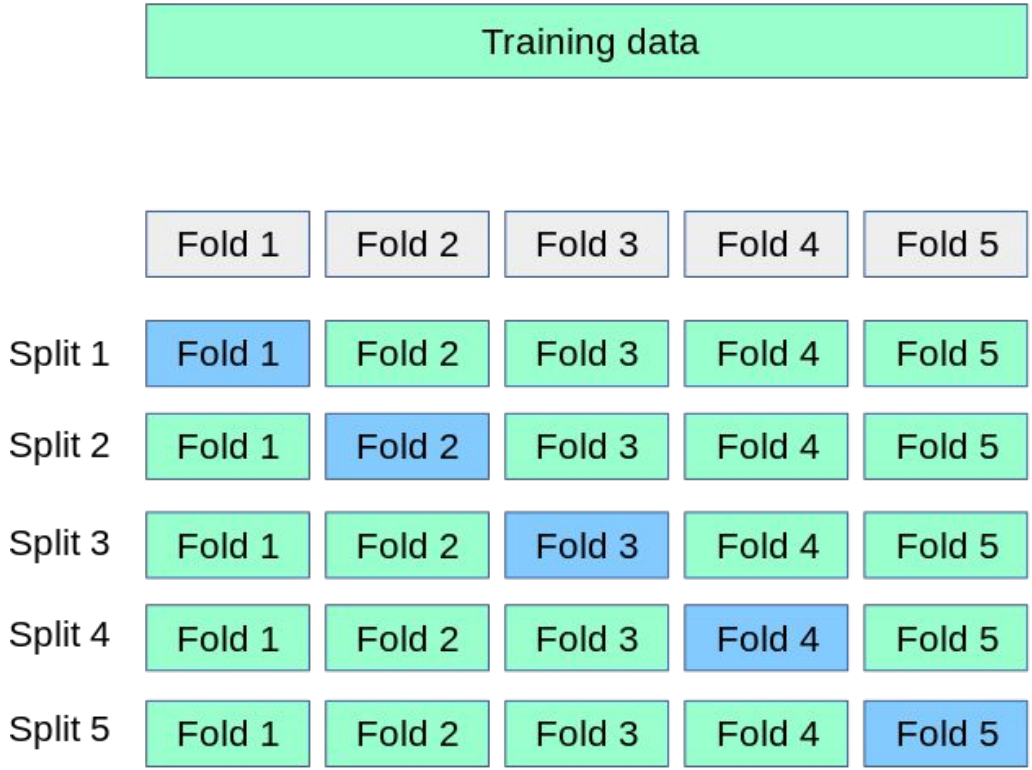


Best model

First order (linear) model generalises best.

Cross-validation (CV)

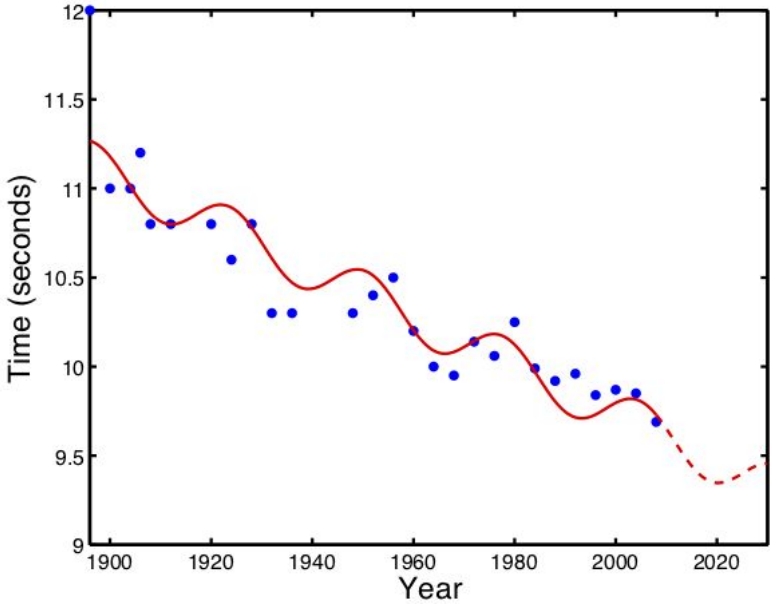
- **Cross-validation can be repeated to make results more accurate.**
- e.g. Doing 5-fold CV 10 times gives us 50 performance values to average over.
- Extreme example is when C = N so each fold includes one input-response pair: **Leave-one-out (LOO) CV**.



$$t = w_0 + w_1 x + w_2 \sin\left(\frac{x-a}{b}\right)$$

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_1 & \sin((x_1-a)/b) \\ \vdots & \vdots & \vdots \\ 1 & x_N & sin((x_N-a)/b) \end{bmatrix}$$

What if you don't like polynomial?



5-fold CV loss at order 3

In [205]:
```
from sklearn.model_selection import KFold
cv = KFold(n_splits = 5)
loss = []
reg = LinearRegression()

poly_order = 3
X_train = make_polynomial(x, poly_order)

for train_index, test_index in cv.split(X_train):
    print('TRAIN:', train_index, 'TEST:', test_index)
    X_train_cv, X_test_cv = X_train[train_index], X_train[test_index]
    t_train_cv, t_test_cv = t[train_index], t[test_index]
    reg.fit(X_train_cv, t_train_cv)
    loss.append( np.mean(( t_test_cv - reg.predict(X_test_cv) )**2  ) )
print(loss)
print(np.mean(loss))
```

```
TRAIN: [ 6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26] TEST:
[0 1 2 3 4 5]
TRAIN: [ 0  1  2  3  4  5 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26] TEST:
[ 6  7  8  9 10 11]
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11 17 18 19 20 21 22 23 24 25 26] TES
T: [12 13 14 15 16]
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 22 23 24 25 26] TES
T: [17 18 19 20 21]
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21] TES
T: [22 23 24 25 26]
[0.08685649261378885, 0.03325048452748726, 0.03685090547650554, 0.008518232044
6148, 0.09683174323737713]
0.052461571579954715
```

5-fold CV loss at order 8

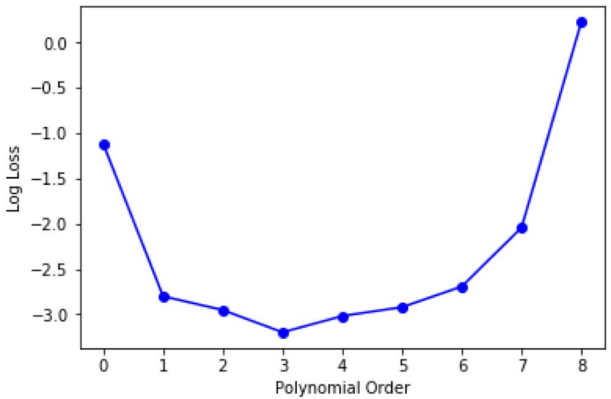
In [206]:

```
poly_order = 8
X_train = make_polynomial(x, poly_order)

for train_index, test_index in cv.split(X_train):
    print('TRAIN:', train_index, 'TEST:', test_index)
    X_train_cv, X_test_cv = X_train[train_index], X_train[test_index]
    t_train_cv, t_test_cv = t[train_index], t[test_index]
    reg.fit(X_train_cv, t_train_cv)
    loss.append( np.mean(( t_test_cv - reg.predict(X_test_cv) )**2  ) )
print(loss)
print(np.mean(loss))
```

```
TRAIN: [ 6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26] TEST:
[0 1 2 3 4 5]
TRAIN: [ 0  1  2  3  4  5 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26] TEST:
[ 6  7  8  9 10 11]
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11 17 18 19 20 21 22 23 24 25 26] TES
T: [12 13 14 15 16]
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 22 23 24 25 26] TES
T: [17 18 19 20 21]
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21] TES
T: [22 23 24 25 26]
[0.08685649261378885, 0.03325048452748726, 0.03685090547650554, 0.008518232044
6148, 0.09683174323737713, 3687.52829345894, 0.22444551630543405, 0.0475836060
2775615, 0.1153987344449662, 107.71191669877541]
379.58899458723937
```

In [221]:
```
cv = KFold(n_splits = 10)
reg = LinearRegression()
all_loss = []
for i in range(9):
    poly_order = i
    X_train = make_polynomial(x, poly_order)
    loss_at_order = []
    for train_index, test_index in cv.split(X_train):
        X_train_cv, X_test_cv = X_train[train_index], X_train[test_index]
        t_train_cv, t_test_cv = t[train_index], t[test_index]
        reg.fit(X_train_cv, t_train_cv)
        loss_at_order.append( np.mean(( t_test_cv - reg.predict(X_test_cv) )**2  )
)
    all_loss.append(np.mean(loss_at_order))
plt.plot(np.log(all_loss), 'bo-')
plt.xlabel('Polynomial Order') # always label x&y-axis
plt.ylabel('Log Loss') # always label x&y-axis
```

Out[221]:
```
Text(0, 0.5, 'Log Loss')
```



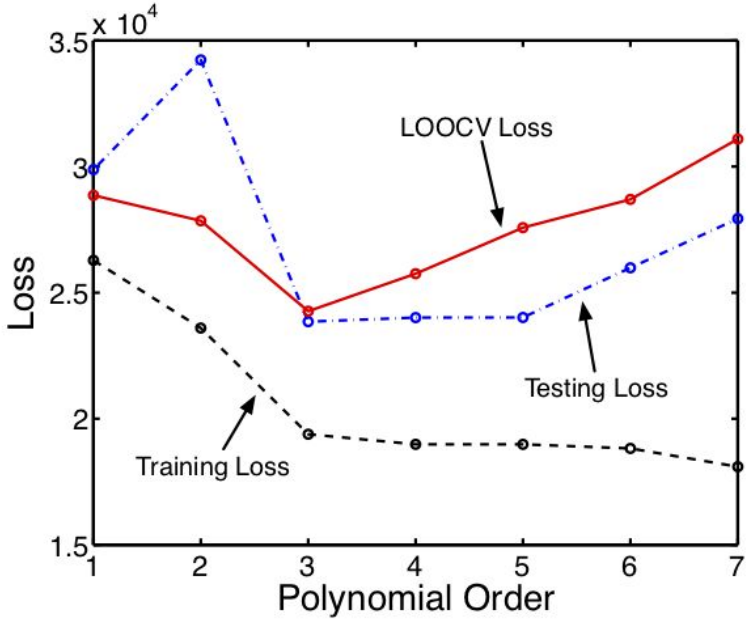
10 fold CV at polynomial order 0 to 8

- Generate some data from a 3rd order model

$$t = w_0 + w_1 x + w_2 x^2 + w_3 x^3.$$

- Use LOOCV to compare models from first to 7th order:

Leave-one-out CV (LOOCV) on a synthetic dataset (We know the right answer!)



(Testing loss comes from another dataset)

General form

$$\mathbf{X} = \begin{bmatrix} h_0(x_1) & h_1(x_1) & \dots & h_K(x_1) \\ h_0(x_2) & h_1(x_2) & \dots & h_K(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ h_0(x_N) & h_1(x_N) & \dots & h_K(x_N) \end{bmatrix}$$

General Linear Regression

$$\widehat{\mathbf{w}} = (\mathbf{X}^{\mathsf{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{t}$$

Where $\mathbf{X}$ depends on the choice of model:

$$\mathbf{X} = \begin{bmatrix} h_0(x_1) & h_1(x_1) & \dots & h_K(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ h_0(x_N) & h_1(x_N) & \dots & h_K(x_N) \end{bmatrix}$$

To predict t at a new value of x, we first create $\mathbf{x}_{\text{new}}$:

$$\mathbf{x}_{\text{new}} = \begin{bmatrix} h_0(x_{\text{new}}) \\ \vdots \\ h_K(x_{\text{new}}) \end{bmatrix},$$

and then compute

$$t_{\text{new}} = \widehat{\mathbf{w}}^{\mathsf{T}}\mathbf{x}_{\text{new}}$$

Common basis functions

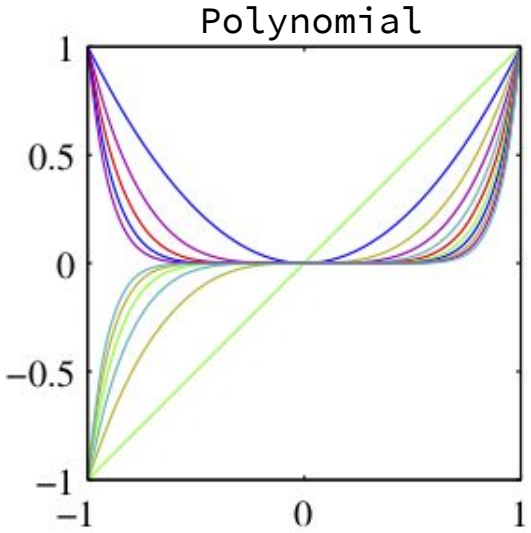
Radial basis function (RBF)

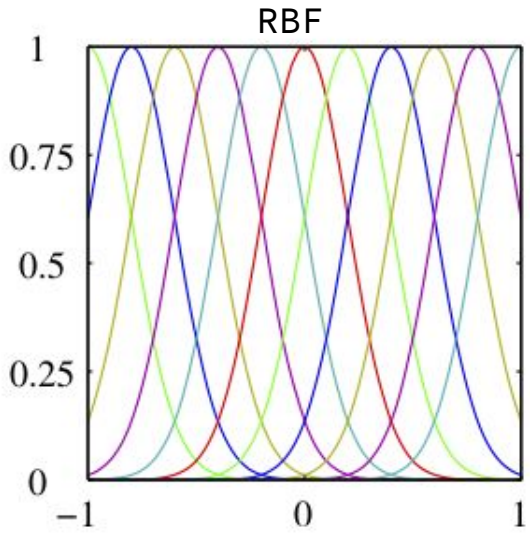
$$h_k(x) = \exp\left(-\frac{(x-\mu_k)^2}{2s^2}\right)$$

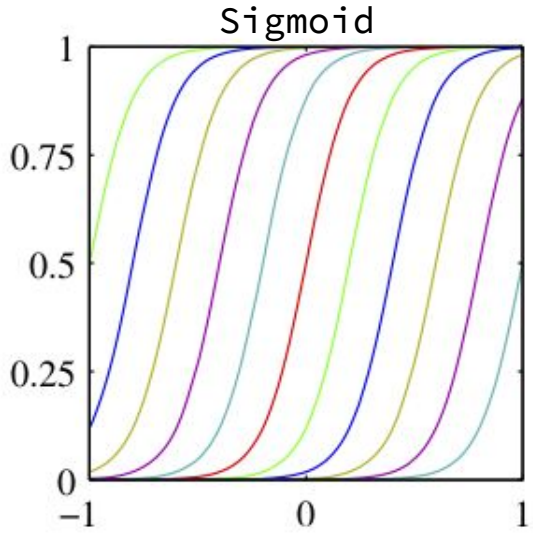
Sigmoid function

$$h_k(x) = \sigma\left(\frac{(x-\mu_k)^2}{s}\right)$$

$$\sigma(a) = \frac{1}{1+\exp(-a)}$$







Summary

- Showed how we can make predictions with our 'linear' model.
- Saw how choice of model has big influence in quality of predictions.
- Saw how the loss on the training data, $\mathcal{L}$, cannot be used to choose models.
  - Making model more complex always decreases the loss.
- Introduced the idea of using some data for validation.
- Introduced cross validation and leave-one-out cross validation.

Machine Learning & Artificial Intelligence for Data Scientists: Regression (Part 3)

Ke Yuan
https://kyuanlab.org/
School of Computing Science

What happens to the parameters when a complex model is overfitting?

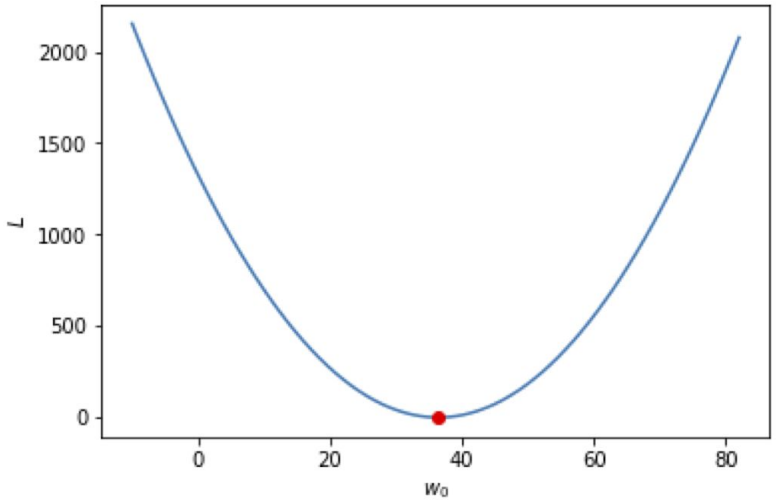
In [234]:
```
for i in range(1, 10, 2):
    poly_order = i
    X_train = make_polynomial(x, poly_order)
    poly_reg = LinearRegression().fit(X_train, t)
    param = poly_reg.coef_[0]
    param[0] = poly_reg.intercept_[0]
    print("order: ", i, ":", param )
```

```
order:  1 : [11.14109659 -0.53323543]
order:  3 : [11.52829666 -1.95586746  1.02257133 -0.19981031]
order:  5 : [11.76096702 -4.53086023  6.99555283 -5.37006803  1.88766668 -0.24
62754 ]
order:  7 : [ 11.88431396  -8.2542977   26.51293334 -44.03421185  38.5909018
 -18.249853     4.40637879  -0.42604153]
order:  9 : [  11.98032237  -15.41186085   86.9871579  -240.58612568  363.1578
9942
 -322.96642614  174.03310952  -55.86414073    9.82843863   -0.72955703]
```

We don’t want the parameters to be too big in absolute value

In [39]:
```
L = np.zeros(num_candidates) # preallocating a vector for L e.g. np.zeros
for j in range(num_candidates): # For loop to evaluate L at every w0_candidates wi
th w1 = -0.013330885711.
    L[j] =  np.mean( (t-w0_candidates[j]-(-0.013330885711)*x)**2 )

plt.plot(w0_candidates, L)
plt.plot(36.4164559025,0.05, 'ro')
plt.xlabel('$w_0$')
plt.ylabel('$L$')
```

Out[39]:
```
Text(0, 0.5, '$L$')
```

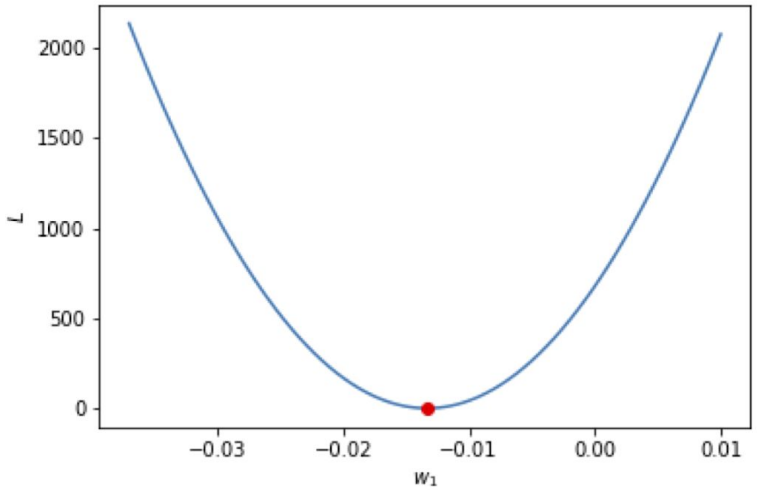


Let's look at the loss function in 1D

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{N}(\mathbf{t} - \mathbf{X}\mathbf{w})^T(\mathbf{t} - \mathbf{X}\mathbf{w})$$

In [40]:
```
L = np.zeros(num_candidates) # preallocating a vector for L e.g. np.zeros
for j in range(num_candidates): # For loop to evaluate L at every w1_candidates wi
th w0 = 36.4164559025
    L[j] =  np.mean( (t-36.4164559025-w1_candidates[j]*x)**2 ) # You can use the
 "np.mean" function

plt.plot(w1_candidates, L) # plot the resulting
plt.plot(-0.013330885711,0.05, 'ro')
plt.xlabel('$w_1$')
plt.ylabel('$L$')
```

Out[40]:
```
Text(0, 0.5, '$L$')
```

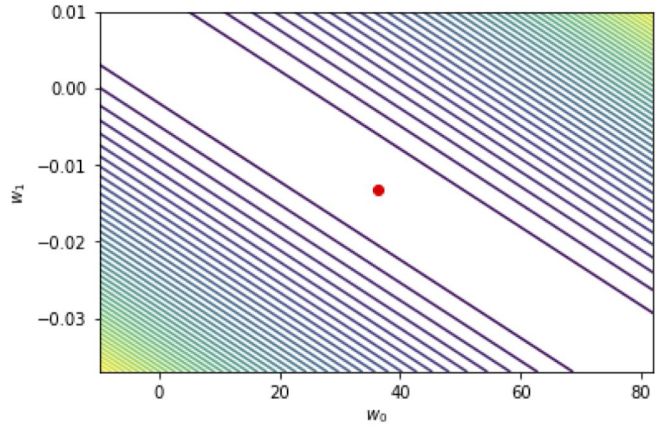


Let's look at the loss function in 1D

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{N}(\mathbf{t} - \mathbf{X}\mathbf{w})^T(\mathbf{t} - \mathbf{X}\mathbf{w})$$

In [41]:
```
L = np.zeros( shape = (num_candidates,num_candidates) ) # Prelocate the loss. We a
re going to have num_candidates times num_candidates of them

# Two nested for loops
for i in range(num_candidates):
    for j in range(num_candidates):
        L[i,j] = np.mean( (t-w0_candidates[i]-w1_candidates[j]*x)**2 )


X, Y = np.meshgrid(w0_candidates, w1_candidates) # Make the x and y coordinates fo
r contour plot
plt.contour(X, Y, L, 50)
plt.plot(36.4164559025, -0.013330885711,'ro')
plt.xlabel('$w_0$')
plt.ylabel('$w_1$')
```

Out[41]:
```
Text(0, 0.5, '$w_1$')
```

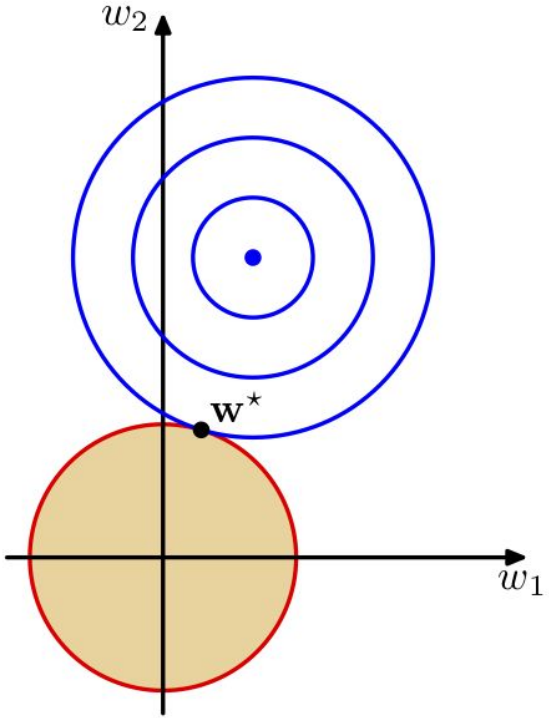


And in 2D

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{N}(\mathbf{t} - \mathbf{X}\mathbf{w})^T(\mathbf{t} - \mathbf{X}\mathbf{w})$$

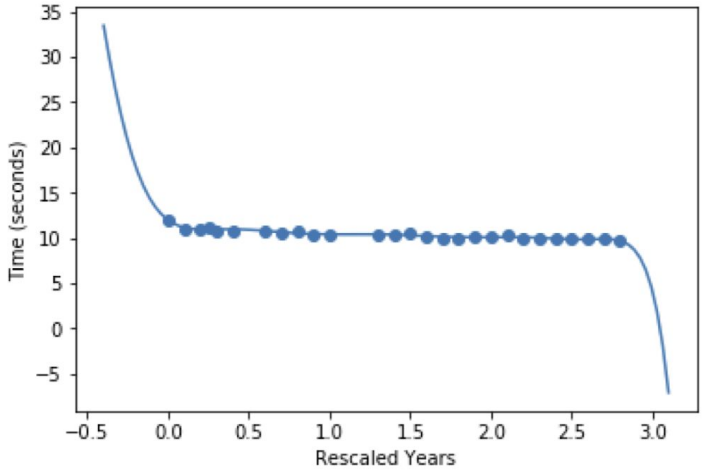
Regularised linear regression: Ridge regression

$$\hat{\mathbf{w}}_{ridge} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{N}(\mathbf{t} - \mathbf{X}\mathbf{w})^T(\mathbf{t} - \mathbf{X}\mathbf{w}) + \alpha \mathbf{w}^T\mathbf{w}$$



In [87]:
```
poly_order = 10
X_train = make_polynomial(x, poly_order)
X_test = make_polynomial(x_test, poly_order)

reg = LinearRegression()
reg.fit(X_train, t)

plt.plot(x_test, reg.predict(X_test))
plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Rescaled Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

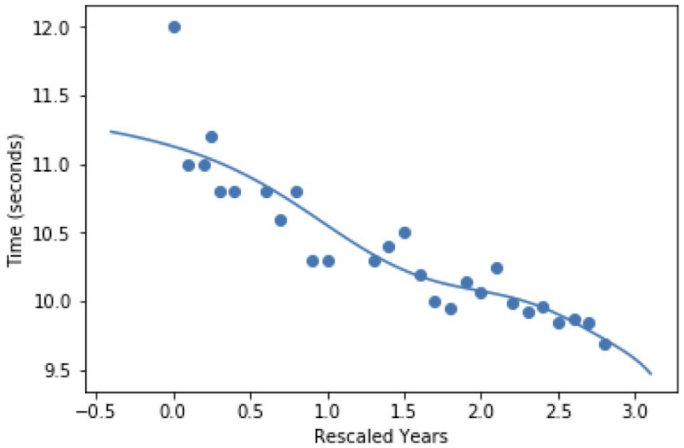
Out[87]:
```
Text(0, 0.5, 'Time (seconds)')
```



Fit a polynomial regression model of order 10

In [88]:
```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

ridge = Ridge()
parameters = {'alpha': np.linspace(1, 10, 20)}
ridge_model = GridSearchCV(ridge, parameters, scoring = 'neg_mean_squared_error',
cv=5)
ridge_model.fit(X_train, t)

plt.plot(x_test, ridge_model.predict(X_test))
plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Rescaled Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

Out[88]:
```
Text(0, 0.5, 'Time (seconds)')
```



Fit a ridge regression model with \alpha determined by 5-fold CV

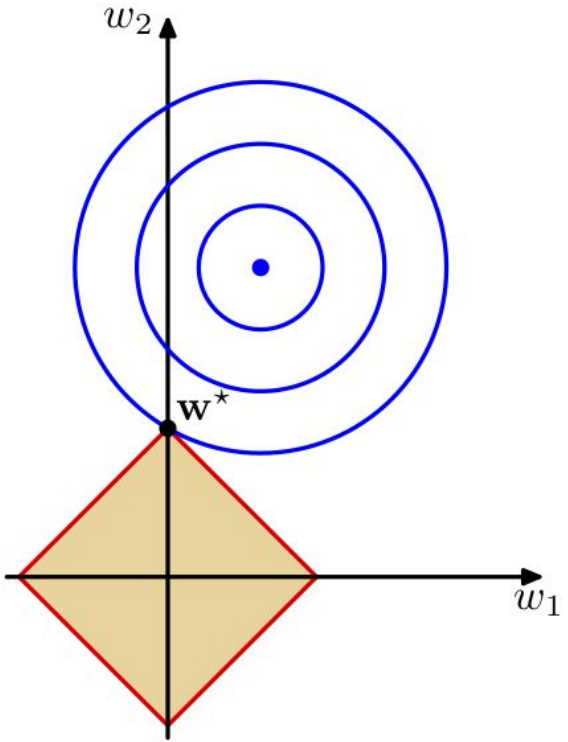
Compare parameters between linear and ridge regression

In [89]:
```
param_lr = reg.coef_[0]
param_ridge = ridge_model.best_estimator_.coef_[0]
param_lr[0] = reg.intercept_[0]
param_ridge[0] = ridge_model.best_estimator_.intercept_[0]
print(np.hstack((param_lr[:,None], param_ridge[:,None])) )
```

```
[[ 1.19648635e+01  1.11285803e+01]
 [-1.29443055e+01 -3.29359603e-01]
 [ 5.79522897e+01 -1.94725736e-01]
 [-1.09582035e+02 -9.64898104e-02]
 [ 6.23248849e+01 -1.87327081e-02]
 [ 7.48519704e+01  3.32402164e-02]
 [-1.46955431e+02  4.50182751e-02]
 [ 1.04735797e+02  9.53751777e-03]
 [-3.88035781e+01 -3.60588365e-02]
 [ 7.43343695e+00  1.40369595e-02]
 [-5.82870289e-01 -1.62830483e-03]]
```

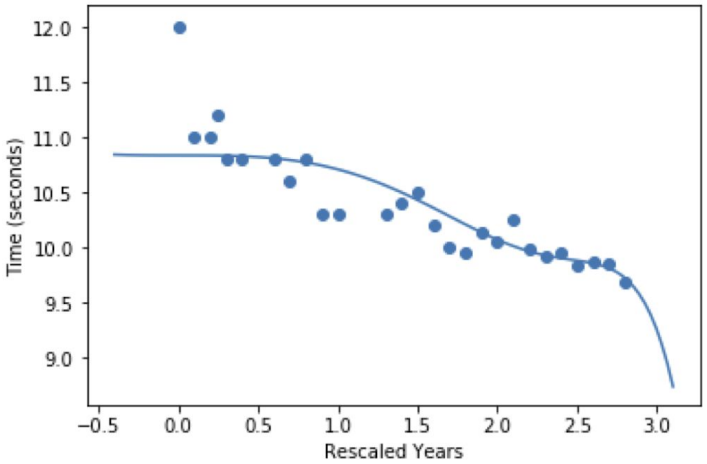
Regularised linear regression: Lasso

$$\hat{\mathbf{w}}_{lasso} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{N}(\mathbf{t} - \mathbf{X}\mathbf{w})^T(\mathbf{t} - \mathbf{X}\mathbf{w}) + \alpha \sum_d |w_d|$$



In [90]:
```
from sklearn.linear_model import Lasso

lasso = Lasso()
parameters = {'alpha': np.linspace(1e-1, 1, 20)}
lasso_model = GridSearchCV(lasso, parameters, scoring = 'neg_mean_squared_error',
cv=5)
lasso_model.fit(X_train, t)

plt.plot(x_test, lasso_model.predict(X_test))
plt.scatter(x,t) # draw a scatter plot
plt.xlabel('Rescaled Years') # always label x&y-axis
plt.ylabel('Time (seconds)') # always label x&y-axis
```

Out[90]:
```
Text(0, 0.5, 'Time (seconds)')
```



Fit a Lasso model with \alpha determined by 5-fold CV

Compare parameters between linear regression, ridge regression, Lasso

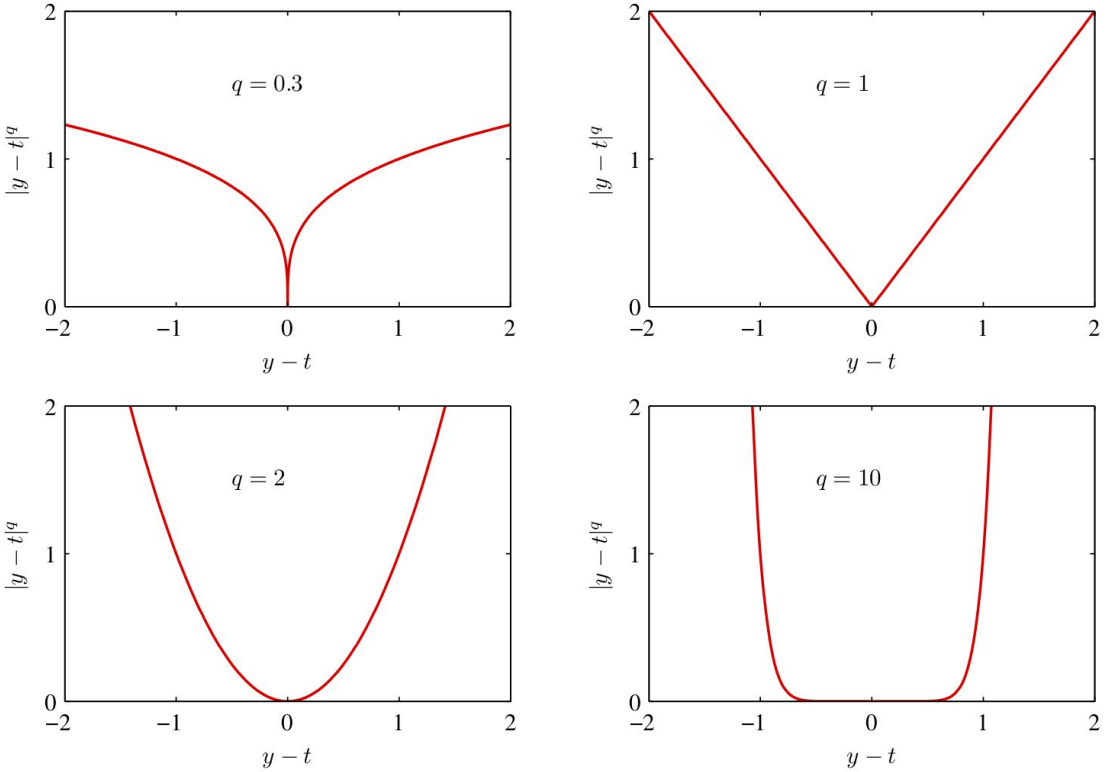
In [91]:
```
param_lasso = lasso_model.best_estimator_.coef_
param_lasso[0] = lasso_model.best_estimator_.intercept_[0]
print(np.hstack((param_lr[:,None], param_ridge[:, None], param_lasso[:,None])) )
```

```
[[ 1.19648635e+01  1.11285803e+01  1.08381535e+01]
 [-1.29443055e+01 -3.29359603e-01 -0.00000000e+00]
 [ 5.79522897e+01 -1.94725736e-01 -0.00000000e+00]
 [-1.09582035e+02 -9.64898104e-02 -1.16126582e-01]
 [ 6.23248849e+01 -1.87327081e-02 -1.59001968e-02]
 [ 7.48519704e+01  3.32402164e-02 -0.00000000e+00]
 [-1.46955431e+02  4.50182751e-02  1.38119952e-03]
 [ 1.04735797e+02  9.53751777e-03  3.22128802e-03]
 [-3.88035781e+01 -3.60588365e-02  1.61616847e-04]
 [ 7.43343695e+00  1.40369595e-02 -8.65262203e-05]
 [-5.82870289e-01 -1.62830483e-03 -7.74413350e-05]]
```

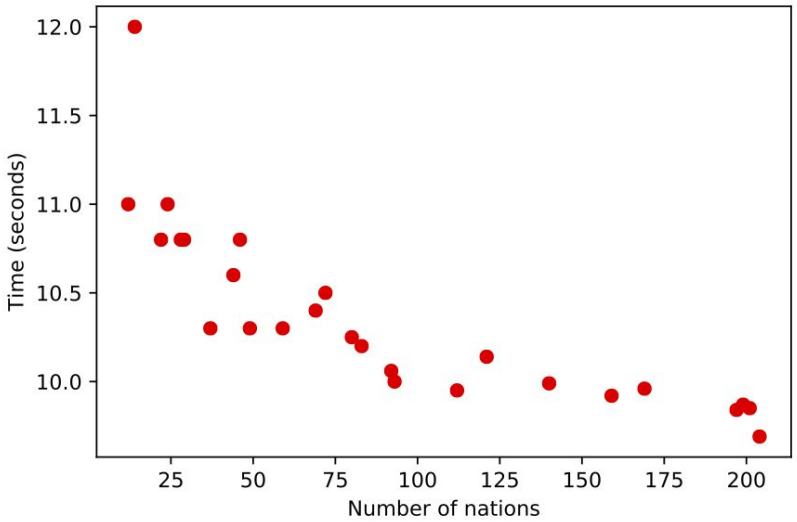
What about a different loss?

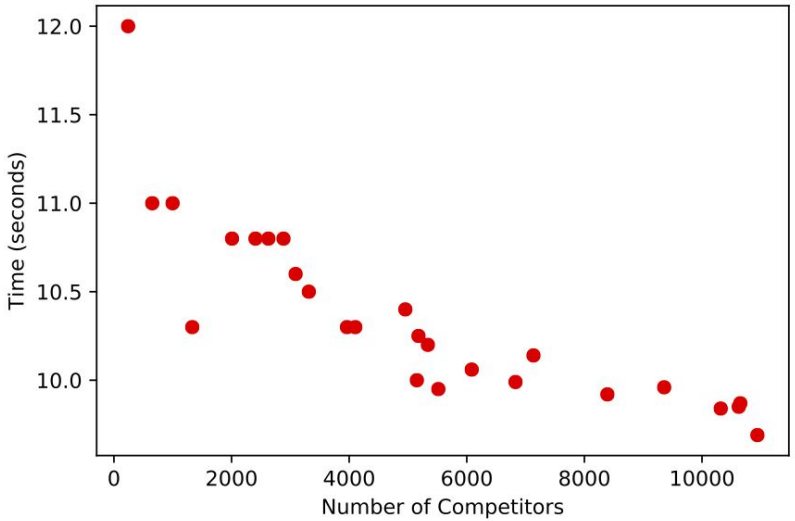
– – –

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{n}^{N} |t_n - \mathbf{w}^T \mathbf{x}_n|^q$$



Should we care about what is x?

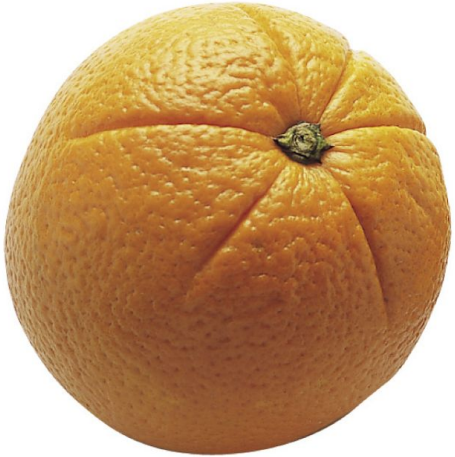




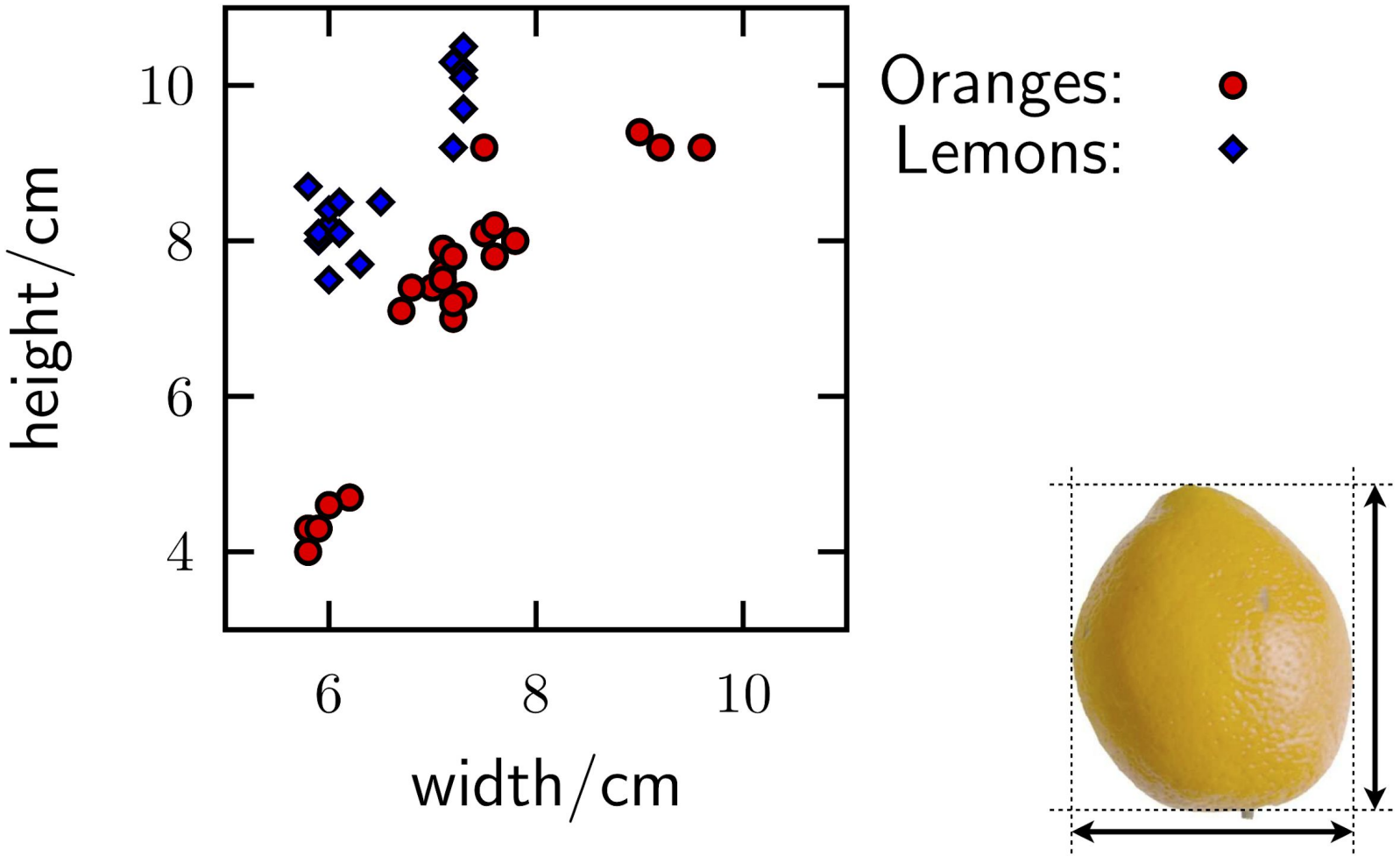
Machine Learning & Artificial Intelligence for Data Scientists: Classification (Part1)

Ke Yuan
https://kyuanlab.org/
School of Computing Science

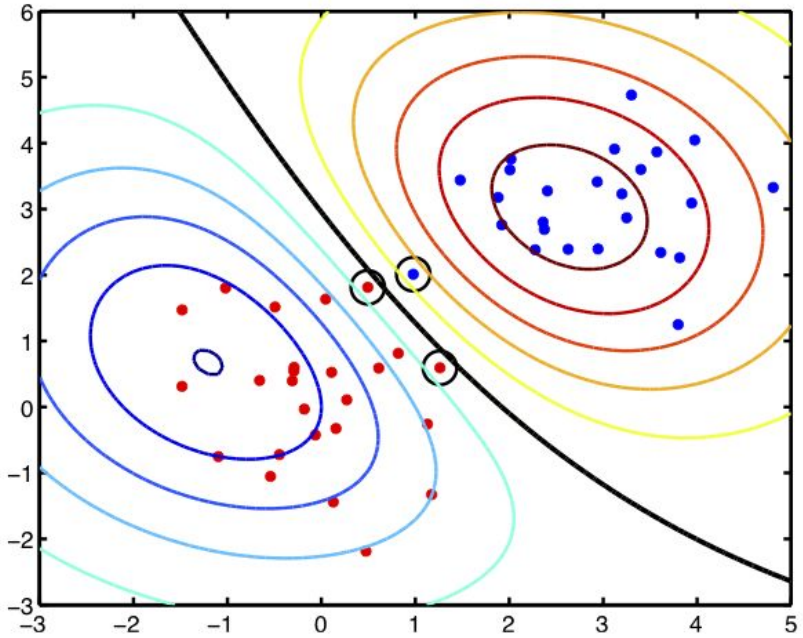
Again some data, and a problem



Source: https://homepages.inf.ed.ac.uk/imurray2/teaching/oranges_and_lemons/


height/cm
10
8
6
4
6
8
10
width/cm
Oranges:
Lemons:

Classification



- A set of N objects with attributes (usually vector) $\mathbf{x}_n$.
- Each object has an associated response (or label) t_n.
- Binary classification: $t_n = \{0, 1\}$ or $t_n = \{-1, 1\}$,
  - (depends on algorithm).
- Multi-class classification: $t_n = \{1, 2, \ldots, K\}$.

Probabilistic v non-probabilistic classifiers

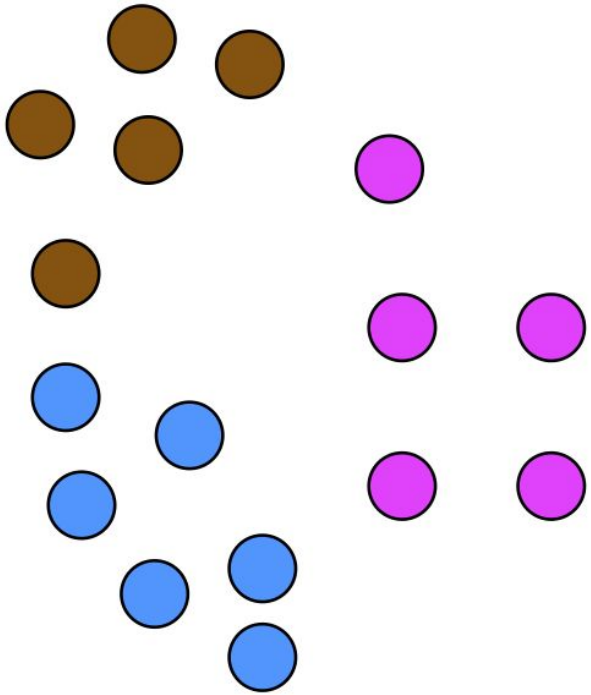
Classifier is trained on $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and $t_1, \ldots, t_N$ and then used to classify $\mathbf{x}_{\text{new}}$.

- Probabilistic classifiers produce a probability of class membership $P(t_{\text{new}} = k | \mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{t})$
  - e.g. binary classification: $P(t_{\text{new}} = 1 | \mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{t})$ and $P(t_{\text{new}} = 0 | \mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{t})$.
- Non-probabilistic classifiers produce a hard assignment
  - e.g. $t_{\text{new}} = 1$ or $t_{\text{new}} = 0$.
- Which to choose depends on application....

Probabilistic v non-probabilistic classifiers

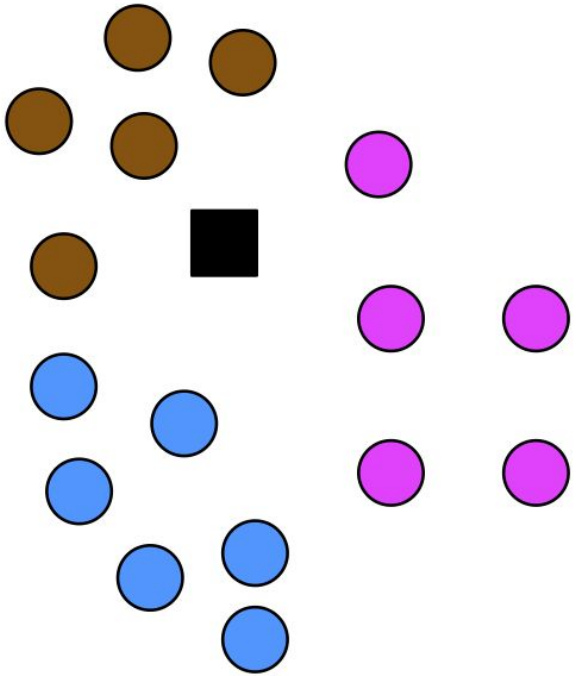
- Probabilities provide us with more information – $P(t_{\text{new}} = 1) = 0.6$ is more useful than $t_{\text{new}} = 1$.
  - Tells us how **sure** the algorithm is.
- Particularly important where cost of misclassification is high and imbalanced.
  - e.g. Diagnosis: telling a diseased person they are healthy is much worse than telling a healthy person they are diseased.
- Extra information (probability) often comes at a cost.

Algorithm 1: K-Nearest Neighbours (KNN)



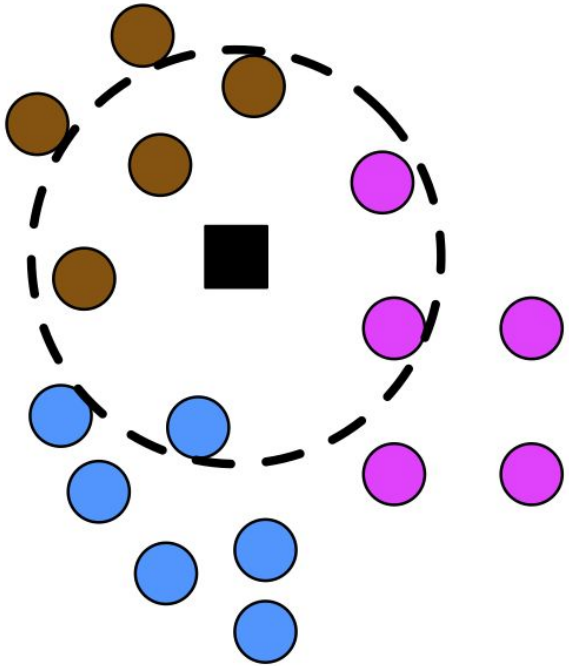
Training data from 3 classes.

KNN



Test point.

KNN



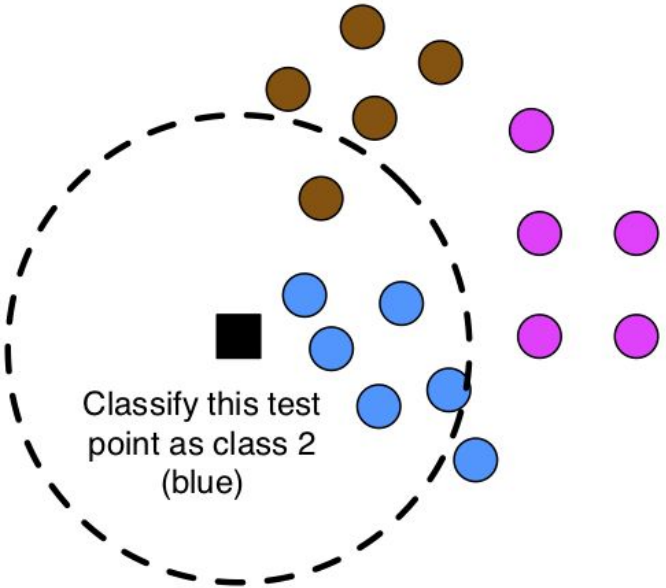
Find $K = 6$ nearest neighbours.

KNN

–

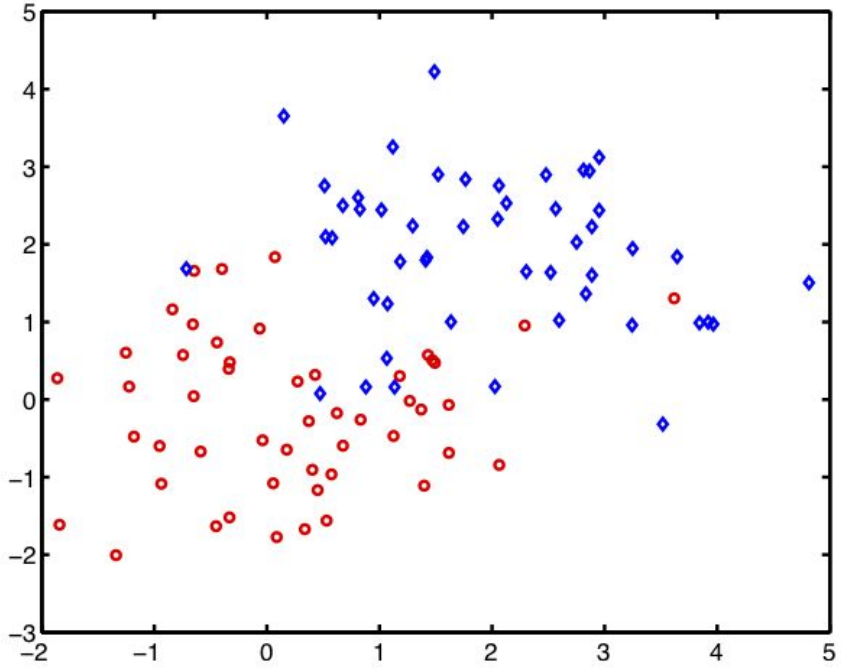
Class one has most votes – classify $\mathbf{x}_{\text{new}}$ as belonging to class 1.

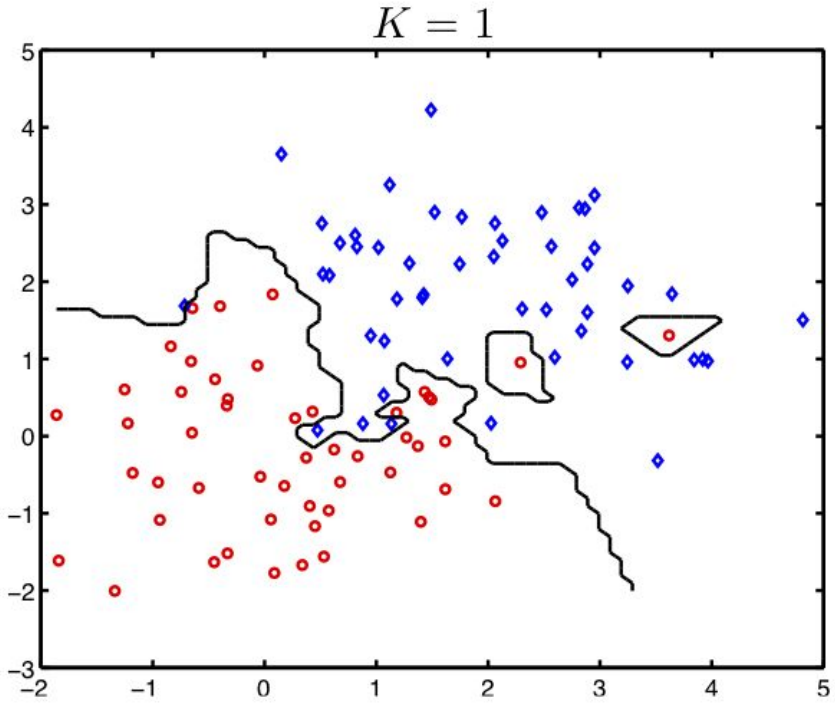
KNN



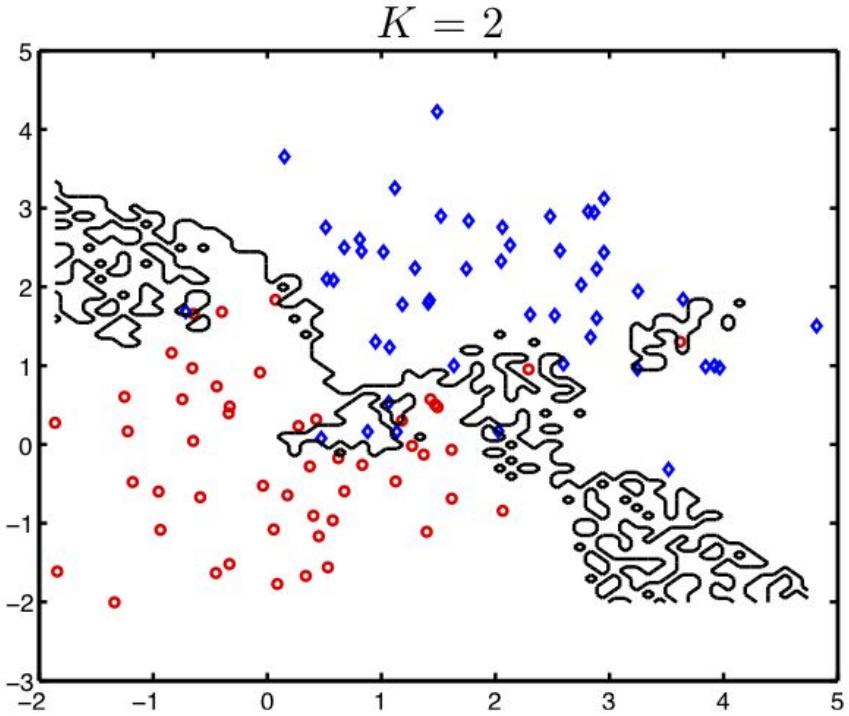
Second example – class 2 has most votes.

KNN: Binary data

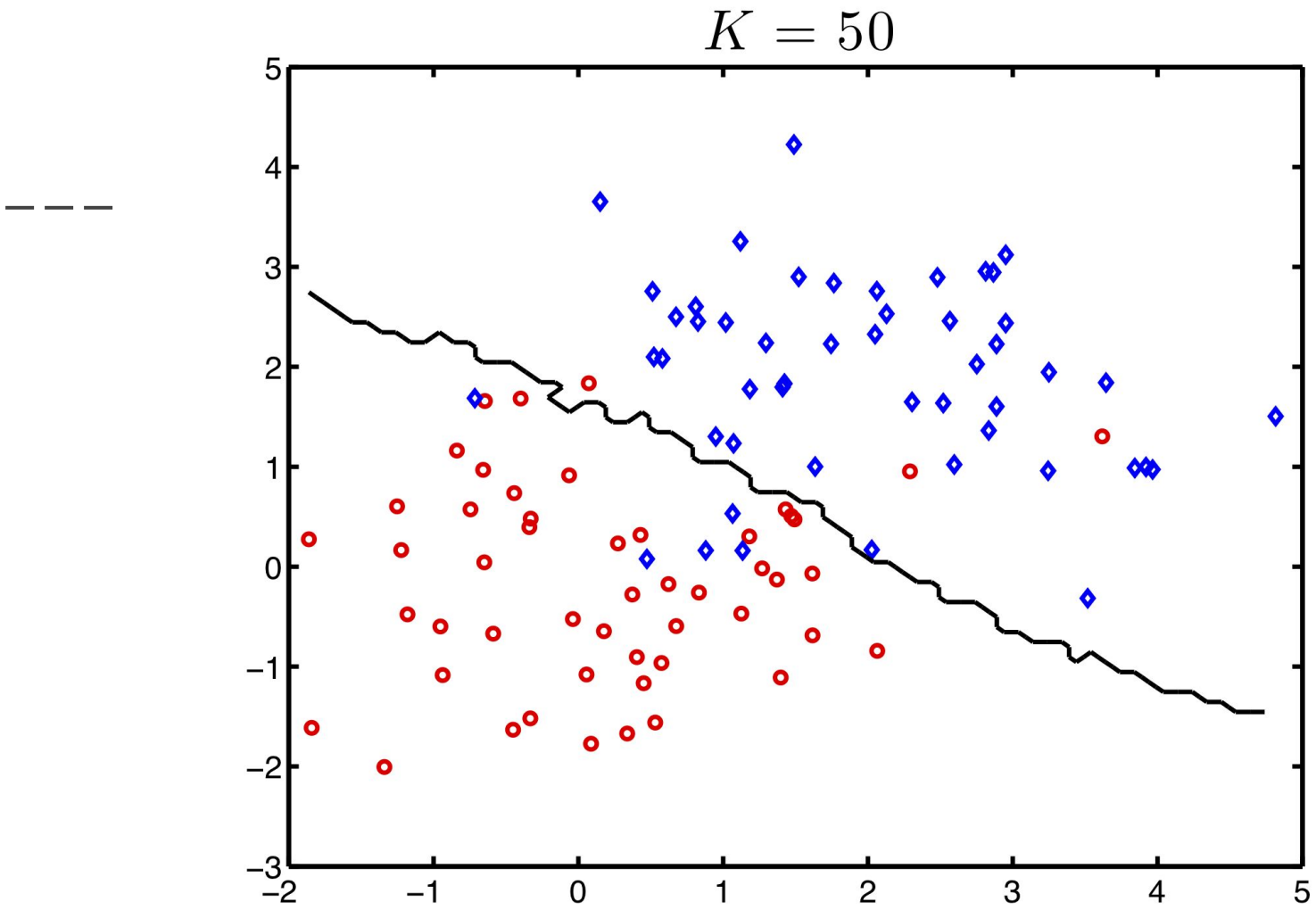




- 1-Nearest Neighbour.
- Line shows decision boundary.
- Too complex – should the islands exist?



- 2-Nearest Neighbour.
- What's going on?
- Lots of ties – random guessing.

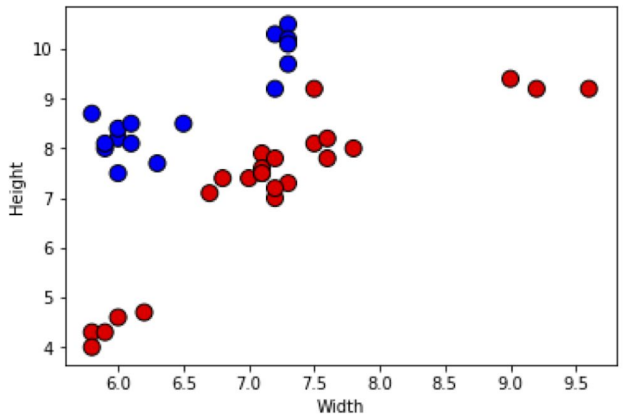

$K = 50$
5
4
3
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
4
5

Problem with KNN

- Class imbalance
  - As K increases, small classes will disappear!
  - Imagine we had only 5 training objects for class 1 and 100 for class 2.
  - For $K \geq 11$, class 2 will **always** win!
- How do we choose K?
  - Right value of K will depend on data.
  - Cross-validation!

In [2]:
```
import numpy as np
%matplotlib inline
import pylab as plt
from matplotlib.colors import ListedColormap

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

data = np.loadtxt('orange_lemon.txt', delimiter=',') # load fruit data
X = data[:,1:3]
t = data[:,0]
plt.scatter(X[:, 0], X[:, 1], c=t, cmap=cmap_bold, edgecolor='k', s=100)
plt.xlabel('Width')
plt.ylabel('Height')
```

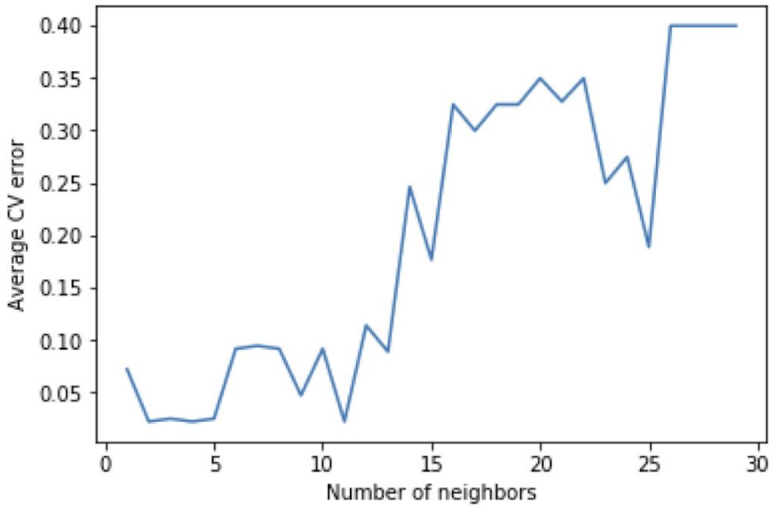
Out[2]:
```
Text(0, 0.5, 'Height')
```



We can see it with oranges and lemons

In [17]:
```
cv_scores = []
for i in range(1,30,1):
    knn_cv = KNeighborsClassifier(n_neighbors=i)
    cv_scores.append(1-np.mean(cross_val_score(knn_cv, X, t, cv=5)))

plt.plot(np.arange(1,30,1),cv_scores)
plt.xlabel('Number of neighbors')
plt.ylabel('Average CV error')
print(np.min(cv_scores))
```

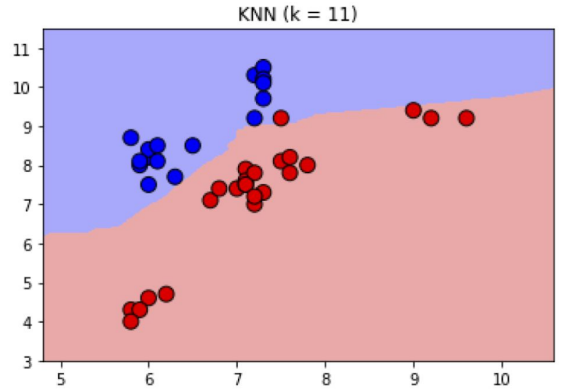
```
0.022222222222222143
```



5-fold CV to select K

In [4]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score

n_neighbors = 11
clf = KNeighborsClassifier(n_neighbors)
clf.fit(X, t)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=t, cmap=cmap_bold,
            edgecolor='k', s=100)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("KNN (k = %i)"
          % (n_neighbors))
```

Out[4]:
```
Text(0.5, 1.0, 'KNN (k = 11)')
```

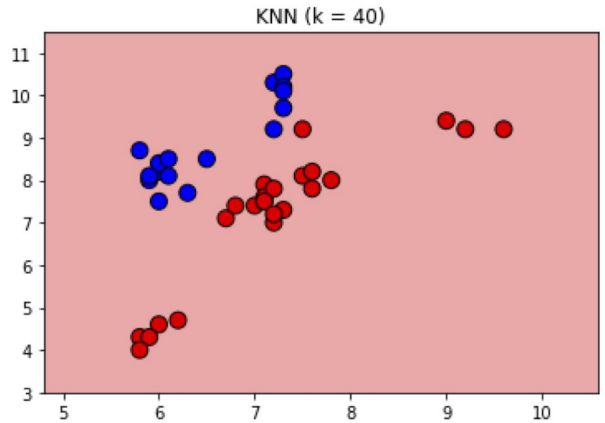


In [3]:
```
# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
h = .02
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
```

K = 11

In [6]:
```
n_neighbors = 40
clf = KNeighborsClassifier(n_neighbors)
clf.fit(X, t)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=t, cmap=cmap_bold,
            edgecolor='k', s=100)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("KNN (k = %i)"
          % (n_neighbors))
```

Out[6]:
```
Text(0.5, 1.0, 'KNN (k = 40)')
```



K = 40

KNN summary

- Non-probabilistic.
- Fast.
- Only one parameter to tune (K).
- Important to tune it well....
- ...can use CV.
- There is a probabilistic version.
  - Not covered in this course.
- Now onto a (different) probabilistic classifier...

Machine Learning & Artificial Intelligence for Data Scientists: Classification (Part2)

Ke Yuan
https://kyuanlab.org/
School of Computing Science

Logistic Regression

- Alternative is to directly model $P(T_{\text{new}} = k|\mathbf{x}_{\text{new}}, \mathbf{X}, \mathbf{t}) = f(\mathbf{x}_{\text{new}}; \mathbf{w})$ with some parameters $\mathbf{w}$.
- We've seen $f(\mathbf{x}_{\text{new}}; \mathbf{w}) = \mathbf{w}^{\mathsf{T}}\mathbf{x}_{\text{new}}$ before – can we use it here?
  - No – *output is unbounded and so can't be a probability.*
- But, can use $P(T_{\text{new}} = k|\mathbf{x}_{\text{new}}, \mathbf{w}) = h(f(\mathbf{x}_{\text{new}}; \mathbf{w}))$ where $h(\cdot)$ *squashes* $f(\mathbf{x}_{\text{new}}; \mathbf{w})$ to lie between 0 and 1 – a probability.

Recap on probability

- Discrete v continuous.
- Probabilities and densities.
- Joint probabilities and densities.
- Independence.
- Conditioning.

Random variables

If I toss a coin and assign the variable X the value 1 if the coin lands heads and 0 if it lands tails, X is a random variable.

We don't know which value X will take but we do know the possible values and how likely they are.

Discrete and continuous RVs

- Random events with outcomes that we can count: Discrete.
  - Coin toss.
  - Rolling a die.
  - Next word in a document.
  - Number of emails sent in a day.

- Random events with outcomes that we cannot count Continuous.
  - Winning time in Olympic 100m.

Discrete and continuous RVs

Definitions

Random variables given capital letters - X, Y.
Lower case letters used for values they can take - x, y.

Discrete RVs

Discrete RVs defines by probabilities of different events taking place. E.g. probability of random variable X taking value x:

$$P(X = x)$$

For example, fair coin:

$$P(X = 1) = 0.5,\ P(X = 0) = 0.5$$

Die:

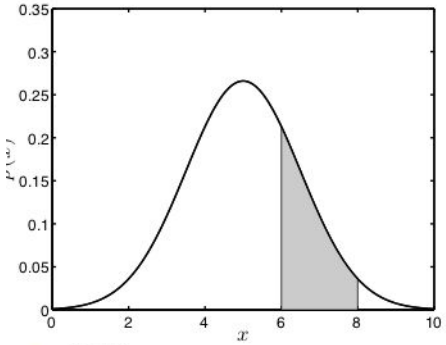
$$P(Y = y) = 1/6$$

Probabilities are constrained:

$$0 \leq P(Y = y) \leq 1,\ \sum_{y} P(Y = y) = 1.$$

Continuous RVs

- Don't define probabilities of particular outcomes as we can't count them!
- Instead define a density function $p(x)$:



$p(x)$ tells us how likely different values are. These are **not** probabilities!

- We can compute probabilities of ranges by computing the area under the curve:

$$P(6 \leq X \leq 8) = \int_{x=6}^{x=8} p(x)\ dx$$

- Densities are constrained:

$$p(x) \geq 0, \quad \int_{-\infty}^{\infty} p(x)\ dx = 1$$

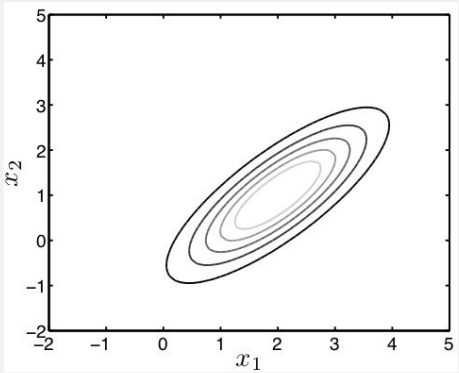
Joint probabilities and densities

Joint probabilities

For two discrete RVs, X and Y, $P(X = x, Y = y)$ is the probability that RV X has value x **and** RV Y has value y.

Joint densities

For two continuous RVs, x_0 and x_1, $p(x_0, x_1)$ is the joint density:



Dependence/Independence

- Let X be the random variable for the toss of a coin (1=heads, 0=tails)
- Let Y be the random variable for the rolling of a die.
- $P(X = 1, Y = 3)$ is the probability that I will roll a head **and** a 3.
- The outcome of X does not depend on Y.
- X and Y are independent.

$$P(X = x, Y = y) = P(X = x)P(Y = y)$$

Dependence/Independence

- Let X be the random variable for the event – I'm playing tennis (1=yes, 0=no)
- Let Y be the random variable for the event – It is raining (1=yes, 0=no)
- $P(X = 1, Y = 1)$ is the probability that I am playing and it is raining.
- The outcome of X **does** depend on Y.
- X and Y are dependent.

$$P(X = x, Y = y) \neq P(X = x)P(Y = y)$$

Conditioning

- Let X be the random variable for the event – I'm playing tennis (1=yes, 0=no)
- Let Y be the random variable for the event – It is raining (1=yes, 0=no)
- Because they are dependent, we can work with **conditional** probabilities.
- e.g. the probability that I am playing **given that** it is raining:

$$P(X = 1|Y = 1)$$

- Allows us to decompose the joint probability:

$$P(X = x, Y = y) = P(X = x|Y = y)P(Y = y)$$

Conditioning - continuous

Example 1:

$$p(t_n|x_n, \mathbf{w})$$

This is the density of t_n conditioned on a particular value of x and our model parameters $\mathbf{w}$.

Example 2:

$$P(9 \leq t_n \leq 10|x_n, \mathbf{w})$$

This is the probability of t_n being between 9 and 10 conditioned on a particular value of x and our model parameters $\mathbf{w}$.

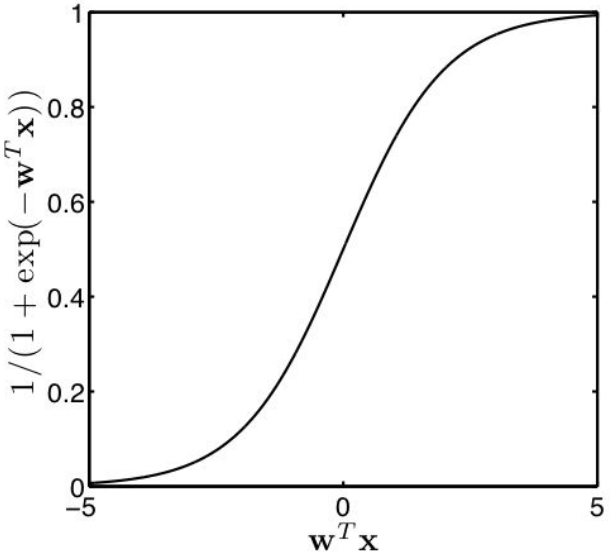
Notational nuance

Technically, we should write $p(t_n|X = x_n, W = \mathbf{w})$ but this becomes unwieldy and gets confusing (difference between X and $\mathbf{X}$?). So, we'll use $p(t_n|x_n, \mathbf{w})$.

Back to logistic regression: Sigmoid function

- For logistic regression (binary), we use the sigmoid function:

$$P(T_{\text{new}} = 1 | \mathbf{x}_{\text{new}}, \mathbf{w}) = h(\mathbf{w}^{\mathsf{T}}\mathbf{x}_{\text{new}}) = \frac{1}{1 + \exp(-\mathbf{w}^{\mathsf{T}}\mathbf{x}_{\text{new}})}$$



Introducing Likelihood of a single label

$$\text{if} \quad t_n = 1, \qquad p(t_n = 1|\mathbf{x}_n, \mathbf{w}) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}_n)}$$

$$\text{if} \quad t_n = 0, \qquad p(t_n = 0|\mathbf{x}_n, \mathbf{w}) = 1 - p(t_n = 1|\mathbf{x}_n, \mathbf{w})$$

One formula for both scenarios

Likelihood function for nth data point

$$p(t_n|\mathbf{x}_n, \mathbf{w}) = p(t_n = 1|\mathbf{x}_n, \mathbf{w})^{t_n}(1 - p(t_n = 1|\mathbf{x}_n, \mathbf{w}))^{(1-t_n)}$$

Likelihood function for all data points

- Assuming data points are independent of each other

$$\text{Likelihood: } p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w})$$

$$\text{Log-Likelihood: } \log p(\mathbf{t}|\mathbf{X}, \mathbf{w}) = \sum_{n=1}^{N} \log p(t_n|\mathbf{x}_n, \mathbf{w})$$

What about Loss?

$$\text{Log-Likelihood}(\mathbf{t}, \mathbf{X}; \mathbf{w}) = \sum_{n=1}^{N} \log p(t_n | \mathbf{x}_n, \mathbf{w})$$

$$\text{Loss}(\mathbf{t}, \mathbf{X}; \mathbf{w}) = -\text{Log-Likelihood}(\mathbf{t}, \mathbf{X}; \mathbf{w}) = -\sum_{n=1}^{N} \log p(t_n | \mathbf{x}_n, \mathbf{w})$$

Find the optimal parameters

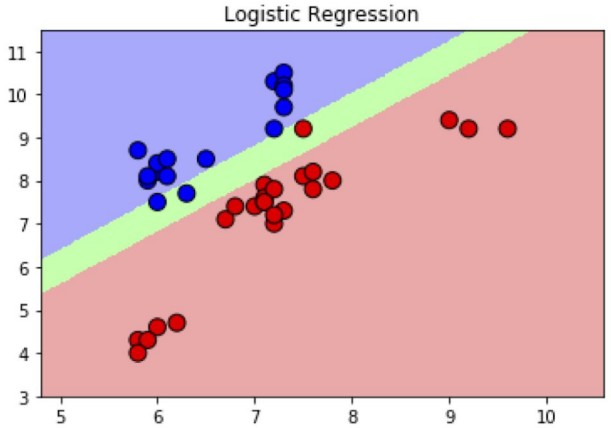
$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}}\ \text{Loss}(\mathbf{t}, \mathbf{X}; \mathbf{w})$$

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmax}}\ \text{Likelihood}(\mathbf{t}, \mathbf{X}; \mathbf{w})$$

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmax}}\ \text{Log-Likelihood}(\mathbf{t}, \mathbf{X}; \mathbf{w})$$

In [59]:
```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression().fit(X, t)
Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
# Plot also the training points
plt.scatter(X[:, 0], X[:, 1], c=t, cmap=cmap_bold,
            edgecolor='k', s=100)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("Logistic Regression")
mean_cv_score = np.mean( cross_val_score(clf, X, t, cv=5) )
print("5-fold averae CV error:", 1-mean_cv_score)
```

```
5-fold averae CV error: 0.025000000000000022
```



Example on orange and lemon data

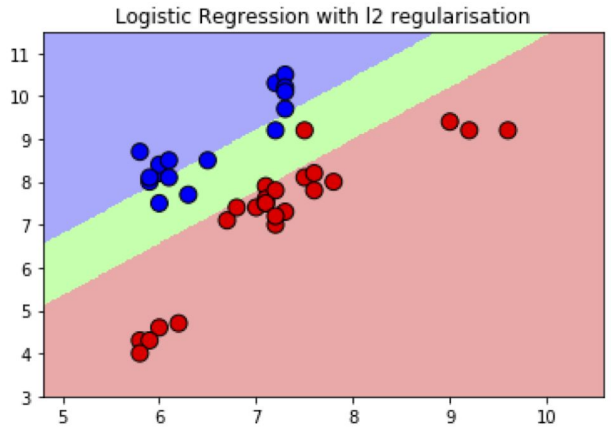
Can be regularised just the same as linear regression

$$\hat{\mathbf{w}}_{l2} = \underset{\mathbf{w}}{\operatorname{argmin}} \operatorname{Loss}(\mathbf{t}, \mathbf{X}; \mathbf{w}) + \frac{1}{C}\mathbf{w}^T\mathbf{w}$$

$$\hat{\mathbf{w}}_{l1} = \underset{\mathbf{w}}{\operatorname{argmin}} \operatorname{Loss}(\mathbf{t}, \mathbf{X}; \mathbf{w}) + \frac{1}{C}\sum_d |w_d|$$

In [53]:
```
parameters = {'C':cs}
logit_reg =  LogisticRegression(penalty='l2', tol=1e-5, max_iter=1e4)
clf = GridSearchCV(logit_reg, parameters, cv=5)
clf.fit(X,t)

Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
Z = Z.reshape(xx.shape) # Put the result into a color plot
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
plt.scatter(X[:, 0], X[:, 1], c=t, cmap=cmap_bold, edgecolor='k', s=100) # Plot al
so the training points
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("Logistic Regression with l2 regularisation")
print("5-fold averae CV error:", 1-clf.best_score_)
```

```
5-fold averae CV error: 0.025000000000000022
```



In [49]:
```
from sklearn.svm import l1_min_c
from sklearn.model_selection import GridSearchCV
cs = l1_min_c(X, t, loss='log')*np.logspace(0, 7, 16)
print(cs)
```

```
[2.89435601e-02 8.47653998e-02 2.48247728e-01 7.27029358e-01
 2.12921058e+00 6.23570098e+00 1.82621518e+01 5.34833516e+01
 1.56633727e+02 4.58724513e+02 1.34344105e+03 3.93446133e+03
 1.15226388e+04 3.37457135e+04 9.88292004e+04 2.89435601e+05]
```

L2 regularised Logistic Regression

Performance Evaluations

- We've seen 2 classification algorithms.
- How do we choose?
  - Which algorithm?
  - Which parameters?
- Need performance indicators.
- We'll cover:
  - 0/1 loss.
  - ROC analysis (sensitivity and specificity)
  - Confusion matrices

0/1 loss

- 0/1 loss: proportion of times classifier is wrong.
- Consider a set of predictions $t_1, \ldots, t_N$ and a set of true labels $t_1^*, \ldots, t_N^*$.
- Mean loss is defined as:

$$\frac{1}{N} \sum_{n=1}^{N} \delta(t_n \neq t_n^*)$$

- ($\delta(a)$ is 1 if a is true and 0 otherwise)
- Advantages:
  - Can do binary or multiclass classification.
  - Simple to compute.
  - Single value.

0/1 loss

Disadvantage: Doesn't take into account class imbalance:

- We're building a classifier to detect a rare disease.
- Assume only 1% of population is diseased.
- Diseased: $t = 1$
- Healthy: $t = 0$
- What if we always predict healthy? $(t = 0)$
- Accuracy 99%
- But classifier is rubbish!

Sensitivity and specificity

- We'll stick with our disease example.
- Need to define 4 quantities. The numbers of:
- True positives (TP) – the number of objects with $t_n^* = 1$ that are classified as $t_n = 1$ (diseased people diagnosed as diseased).
- True negatives (TN) – the number of objects with $t_n^* = 0$ that are classified as $t_n = 0$ (healthy people diagnosed as healthy).
- False positives (FP) – the number of objects with $t_n^* = 0$ that are classified as $t_n = 1$ (healthy people diagnosed as diseased).
- **False negatives (FN)** – the number of objects with $t_n^* = 1$ that are classified as $t_n = 0$ (diseased people diagnosed as healthy).

Sensitivity

$$S_e = \frac{TP}{TP + FN}$$

- The proportion of diseased people that we classify as diseased.
- The higher the better.
- In our example, $S_e = 0$.

Specificity

$$S_p = \frac{TN}{TN + FP}$$

- The proportion of healthy people that we classify as healthy.
- The higher the better.
- In our example, $S_p = 1$.

Optimising sensitivity and specificity

- We would like both to be as high as possible.
- Often increasing one will decrease the other.
- Balance will depend on application:
- e.g. diagnosis:
  - We can probably tolerate a decrease in specificity (healthy people diagnosed as diseased)....
  - ...if it gives us an increase in sensitivity (getting diseased people right).

Receiver Operating Characteristic (ROC)

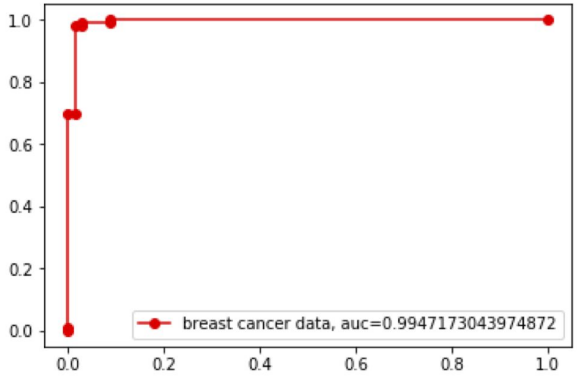
- Many classification algorithms involve setting a threshold.
- e.g. Logistic Regression:

$$p(t_{new} = 1 | \mathbf{x}_{new}, \mathbf{w}) > 0.5$$

- Implies a threshold of zero (sign function)
- However, we could use any threshold we like....
- The *Receiver Operating Characteristic (ROC) curve* shows how S_e and $1 - S_p$ vary as the threshold changes.

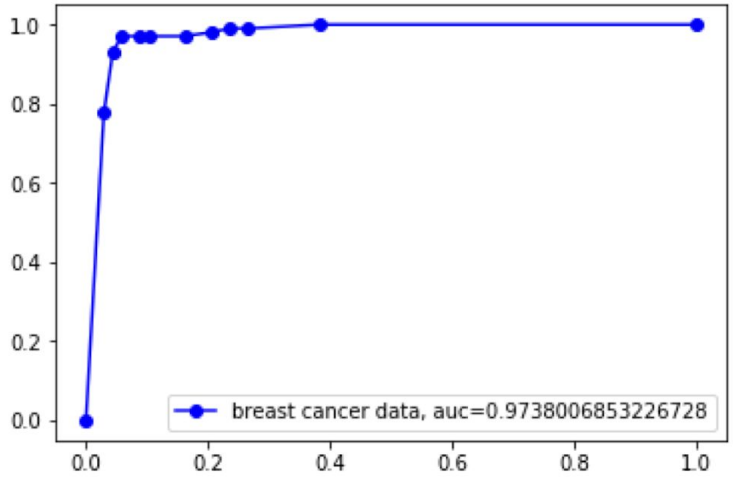
In [16]:

```
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.datasets import load_breast_cancer

breast_cancer = load_breast_cancer()
X = breast_cancer.data
t = breast_cancer.target


X_train, X_test, y_train, y_test = train_test_split(X,t,test_size=0.30, random_sta
te=123)
clf1 = LogisticRegression().fit(X_train, y_train)

y_pred1 = clf1.predict(X_test)
y_pred_proba1 = clf1.predict_proba(X_test)[:,1]
fpr1, tpr1, _ = metrics.roc_curve(y_test,  y_pred_proba1)
auc1 = metrics.roc_auc_score(y_test, y_pred_proba1)
plt.plot(fpr1,tpr1,'ro-',label="breast cancer data, auc="+str(auc1))
plt.legend(loc=4)
plt.show()
```



Try it on a breast cancer dataset
Plot ROC of a Logistic Regression model

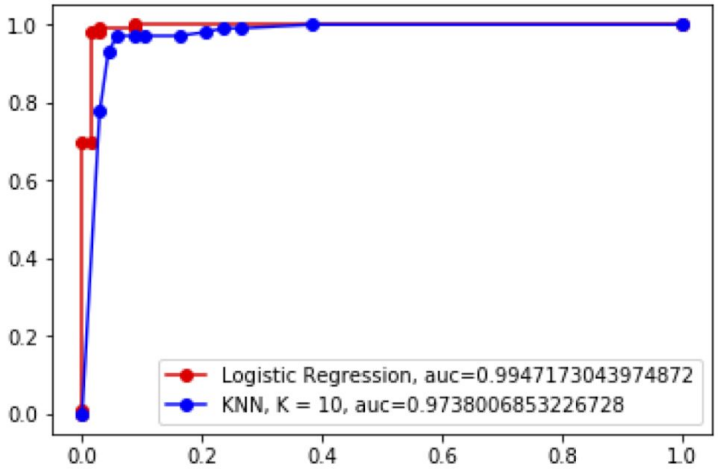
In [19]:
```
clf2 = KNeighborsClassifier(10).fit(X_train, y_train)

y_pred2 = clf2.predict(X_test)
y_pred_proba2 = clf2.predict_proba(X_test)[:,1]
fpr2, tpr2, _ = metrics.roc_curve(y_test,  y_pred_proba2)
auc2 = metrics.roc_auc_score(y_test, y_pred_proba2)
plt.plot(fpr2,tpr2,'bo-',label="breast cancer data, auc="+str(auc2))
plt.legend(loc=4)
plt.show()
```



The same data with KNN (K =10)

Overlay two ROC plots

In [21]:
```
plt.plot(fpr1,tpr1,'ro-',label="Logistic Regression, auc="+str(auc1))
plt.plot(fpr2,tpr2,'bo-',label="KNN, K = 10, auc="+str(auc2))
plt.legend(loc=4)
plt.show()
```



What is happening?

```
Can I have 7 volunteers?
```

Confusion matrix

The quantities we used to compute S_e and S_p can be neatly summarised in a table:

		True class 1	True class 0
Predicted class	1	TP	FP
	0	FN	TN

- This is known as a *confusion matrix*
- It is particularly useful for multi-class classification.
- Tells us where the mistakes are being made.
- Note that normalising columns gives us S_e and S_p

Confusion matrix, example

- 20 newsgroups data.
- Thousands of documents from 20 classes (newsgroups)

		True class										
	...	10	11	12	13	14	15	16	18	18	19	20
Predicted class												
1	...	4	2	0	2	10	4	7	1	12	7	47
2	...	0	0	4	18	7	8	2	0	1	1	3
3	...	0	0	1	0	1	0	1	0	0	0	0
4	...	1	0	1	28	3	0	0	0	0	0	0
⋮												
16	...	3	2	2	5	17	4	376	3	7	2	**68**
17	...	1	0	9	0	3	1	3	325	3	**95**	19
18	...	2	1	0	2	6	2	1	2	325	4	5
19	...	8	4	8	0	10	21	1	16	19	**185**	7
20	...	0	0	1	0	1	1	2	4	0	1	**92**

- Algorithm is getting 'confused' between classes 20 and 16, and 19 and 17.
  - 17: talk.politics.guns
  - 19: talk.politics.misc
  - 16: talk.religion.misc
  - 20: soc.religion.christian
- Maybe these should be just one class?
- Maybe we need more data in these classes?
- Confusion matrix helps us direct our efforts to improving the classifier.

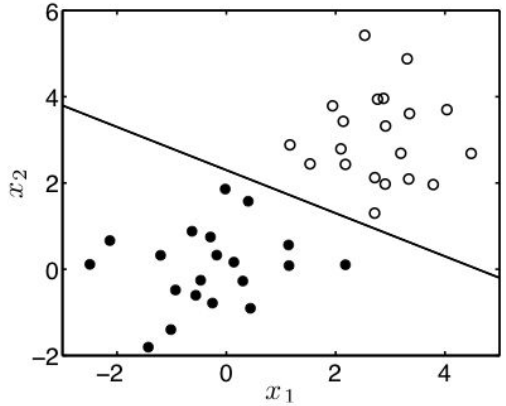
Machine Learning & Artificial Intelligence for Data Scientists: Classification (Part3)

Ke Yuan
https://kyuanlab.org/
School of Computing Science

Support Vector Machines (SVM)

- We have seen two algorithms where we find the parameters that optimise something:
  - Minimise the loss
  - Maximise the likelihood
- The Support Vector Machine (SVM) is no different:
  - **It finds the decision boundary that maximises the margin.**

Some toy data



SVM is a binary classifier. N data points, each with attributes $\mathbf{x} = [x_1, x_2]^\mathsf{T}$ and target $t = \pm 1$

- A linear *decision boundary* can be represented as a straight line:

$$\mathbf{w}^\mathsf{T}\mathbf{x} + b = 0$$

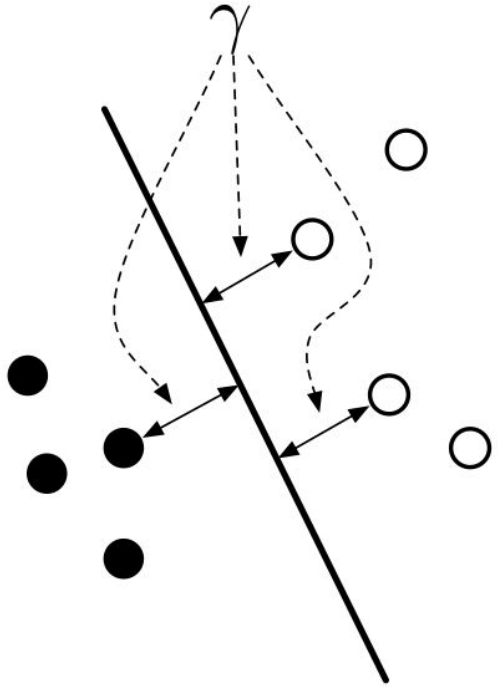
- Our task is to find $\mathbf{w}$ and b
- Once we have these, classification is easy:

$$\mathbf{w}^\mathsf{T}\mathbf{x}_{\text{new}} + b > 0 \quad : \quad t_{\text{new}} = 1$$
$$\mathbf{w}^\mathsf{T}\mathbf{x}_{\text{new}} + b < 0 \quad : \quad t_{\text{new}} = -1$$

- i.e. $t_{\text{new}} = \text{sign}(\mathbf{w}^\mathsf{T}\mathbf{x}_{\text{new}} + b)$

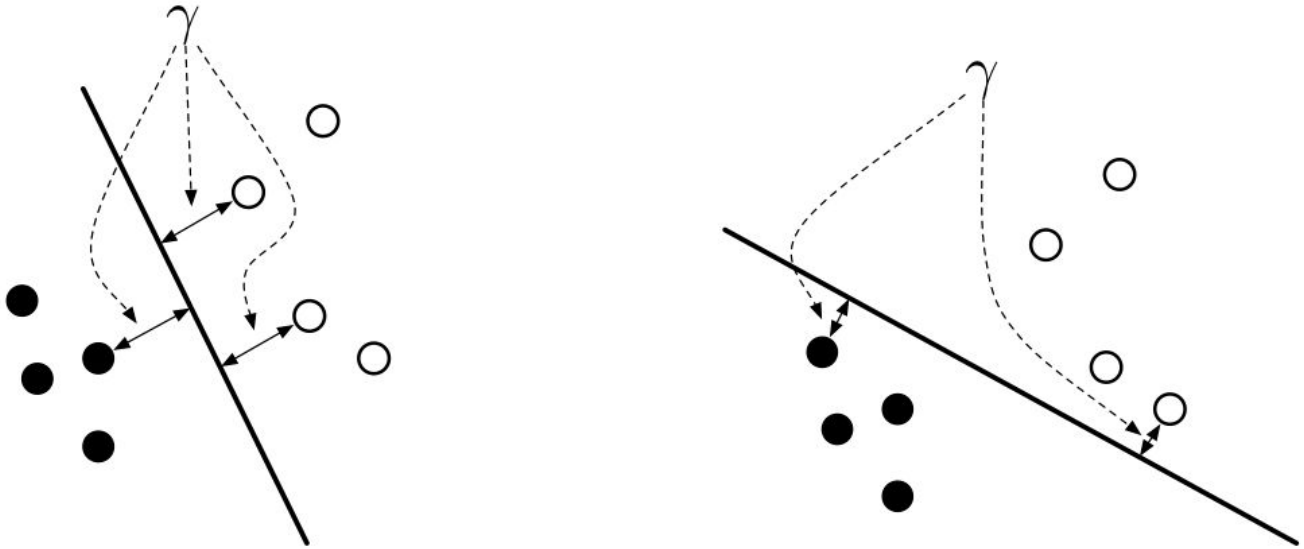
Margin

- How do we choose **w** and b?
- Need a quantity to optimise!
- Use the **margin**, γ
- Maximise it!



Perpendicular distance from the decision boundary to the closest points on each side.

Why maximise the margin?



- Maximum margin decision boundary (left) seems to better reflect the data characteristics than other boundary (right).
- Note how margin is much smaller on right and closest points have changed.
- There is going to be one 'best' boundary (w.r.t margin)

Computing the margin

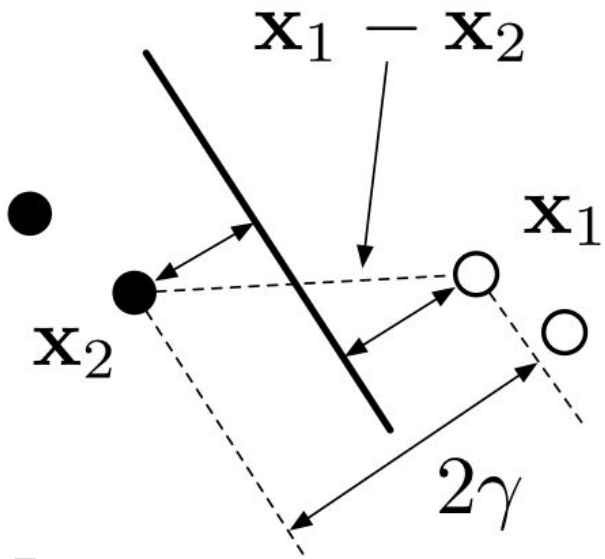
$$2\gamma = \frac{1}{||\mathbf{w}||}\mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2)$$

Fix the scale such that:

$$\begin{aligned} \mathbf{w}^T\mathbf{x}_1 + b &= 1 \\ \mathbf{w}^T\mathbf{x}_2 + b &= -1 \end{aligned}$$

Therefore:

$$\begin{aligned} (\mathbf{w}^T\mathbf{x}_1 + b) - (\mathbf{w}^T\mathbf{x}_2 + b) &= \\ \mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2) &= \\ \gamma = \frac{1}{||\mathbf{w}||} & \end{aligned}$$



Maximising the margin

- We want to maximise $\gamma = \frac{1}{||\mathbf{w}||}$
- Equivalent to minimising $||\mathbf{w}||$
- Equivalent to minimising $\frac{1}{2}||\mathbf{w}||^2 = \frac{1}{2}\mathbf{w}^\mathsf{T}\mathbf{w}$

- There are some constraints:
  - For $\mathbf{x}_n$ with $t_n = 1$: $\mathbf{w}^\mathsf{T}\mathbf{x}_n + b \geq 1$
  - For $\mathbf{x}_n$ with $t_n = -1$: $\mathbf{w}^\mathsf{T}\mathbf{x}_n + b \leq -1$
- Which can be expressed more neatly as:

$$t_n(\mathbf{w}^\mathsf{T}\mathbf{x}_n + b) \geq 1$$

- (This is why we use $t_n = \pm 1$ and not $t_n = \{0, 1\}$.)

Maximising the margin

- We have the following optimisation problem:

$$\underset{\mathbf{w}}{\text{argmin}} \ \frac{1}{2}\mathbf{w}^{\mathsf{T}}\mathbf{w}$$

$$\text{Subject to: } t_n(\mathbf{w}^{\mathsf{T}}\mathbf{x}_n + b) \geq 1$$

- Can put the constraints into the minimisation using *Lagrange multipliers*:

$$\underset{\mathbf{w}}{\text{argmin}} \ \frac{1}{2}\mathbf{w}^{\mathsf{T}}\mathbf{w} - \sum_{n=1}^{N} \alpha_n(t_n(\mathbf{w}^{\mathsf{T}}\mathbf{x}_n + b) - 1)$$

$$\text{Subject to: } \alpha_n \geq 0$$

The final formula

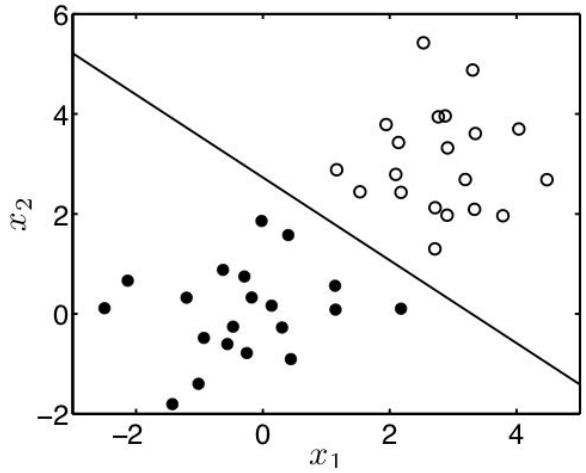
$$\underset{\boldsymbol{\alpha}}{\text{argmax}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n.m=1}^{N} \alpha_n \alpha_m t_n t_m \mathbf{x}_n^{\mathsf{T}} \mathbf{x}_m$$

$$\text{subject to} \quad \sum_{n=1}^{N} \alpha_n t_n = 0, \quad \alpha_n \geq 0$$

- This is a standard optimisation problem (quadratic programming)
- Has a single, global solution. This is very useful!
- Many algorithms around to solve it.
- e.g. `quadprog` in Matlab...
- Once we have α_n:

$$t_{\text{new}} = \text{sign}\left(\sum_{n=1}^{N} \alpha_n t_n \mathbf{x}_n^{\mathsf{T}} \mathbf{x}_{\text{new}} + b \right)$$

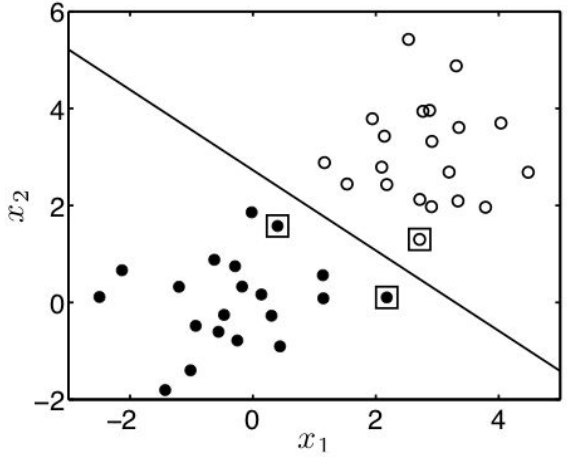
Optimal boundary



- Optimisation gives us $\alpha_1, \ldots, \alpha_N$
- Compute $\mathbf{w} = \sum_n \alpha_n t_n \mathbf{x}_n$
- Compute $b = t_n - \mathbf{w}^\mathsf{T}\mathbf{x}$ (for one of the closest points)
  - Recall that we defined $\mathbf{w}^\mathsf{T}\mathbf{x} + b = \pm 1 = t_n$ for closest points.
- Plot $\mathbf{w}^\mathsf{T}\mathbf{x} + b = 0$

Support Vectors

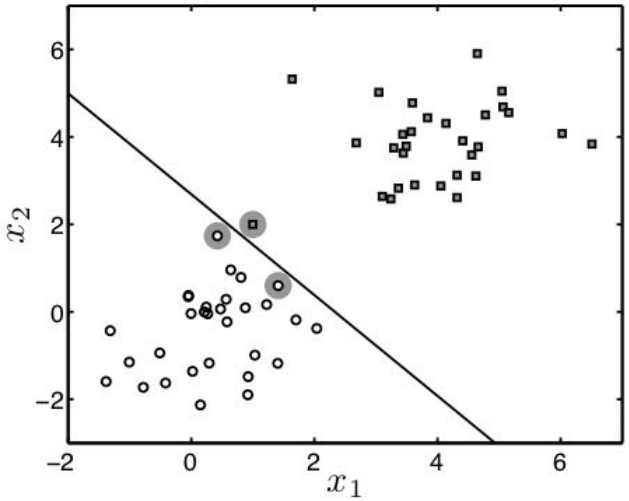
- At the optimum, only 3 non-zero α values (squares).



- $t_{\text{new}} = \text{sign}\left(\sum_n \alpha_n t_n \mathbf{x}_n^{\mathsf{T}} \mathbf{x}_{\text{new}} + b\right)$
- Predictions only depend on these data-points!
- We knew that – margin is only a function of closest points.
- These are called **Support Vectors**
- Normally a small proportion of the data:
  - Solution is *sparse*.

Is sparseness good?

- Not always:



- Why does this happen?

$$t_n(\mathbf{w}^\mathsf{T}\mathbf{x}_n + b) \geq 1$$

- All points must be on correct side of boundary.
- This is a *hard margin*

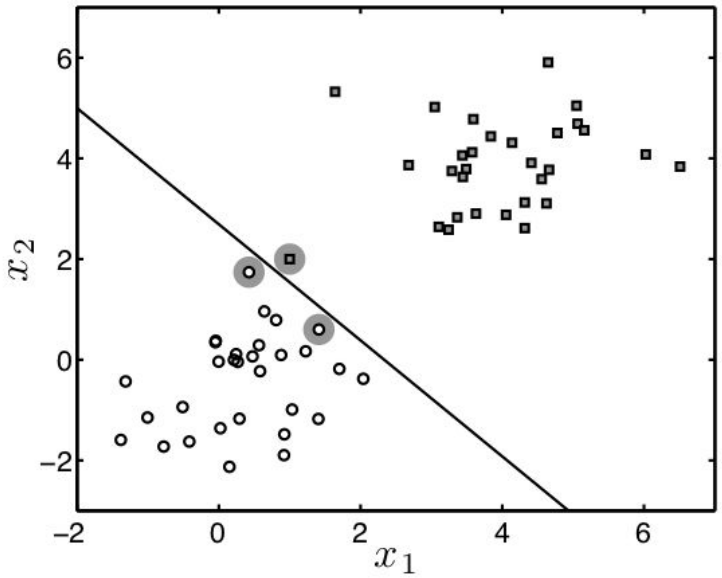
Soft margins

$$\underset{\alpha}{\text{argmax}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n.m=1}^{N} \alpha_n \alpha_m t_n t_m \mathbf{x}_n^{\mathsf{T}} \mathbf{x}_m$$

$$\text{subject to} \quad \sum_{n=1}^{N} \alpha_n t_n = 0, \quad \boxed{0 \leq \alpha_n \leq C}$$

Soft margins

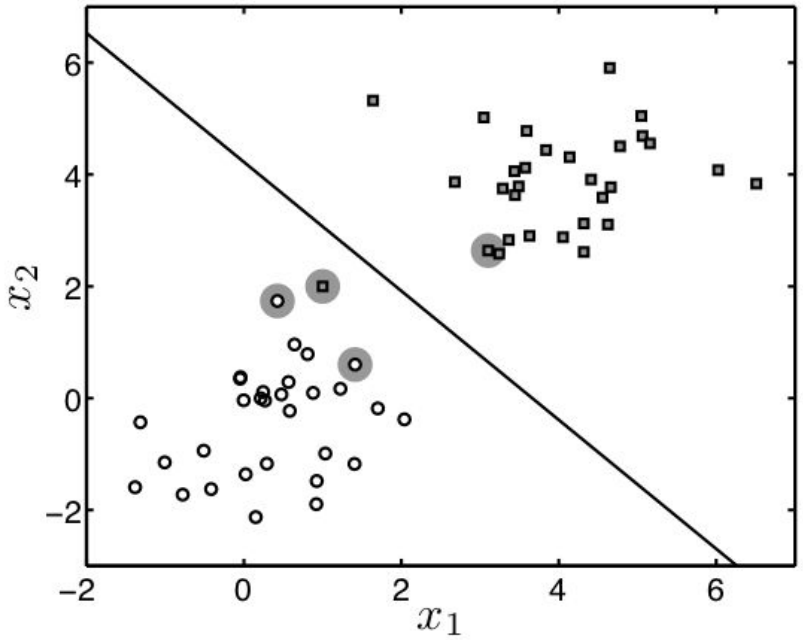
- Here's our problematic data again:



- α_n for the 'bad' square is 3.5.
- So, if we set $C < 3.5$, we should see this point having less influence and the boundary moving to somewhere more sensible...

Soft margins

- Try $C = 1$



- We have an extra support vector.
- And a better decision boundary.

Soft margins

– – –

- The choice of C is very important.
- Too high and we *over-fit* to noise.
- Too low and we *underfit*
  - ...and lose any sparsity.

- Choose it using cross-validation.

SVM - more observations

- In our example, we started with 3 parameters:

$$\mathbf{w} = [w_1, w_2]^{\mathsf{T}}, \quad b$$

- In general: D+1.
- We now have N: $\alpha_1, \ldots, \alpha_N$
- Sounds harder?
- Depends on data dimensionality:
  - Typical Microarray dataset:
  - $D \sim 3000, N \sim 30$.
  - In some cases $N \ll D$

Inner product

- Here's the optimisation problem:

$$\underset{\boldsymbol{\alpha}}{\text{argmax}} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m} \alpha_n \alpha_m t_n t_m \mathbf{x}_n^\mathsf{T} \mathbf{x}_m$$

- Here's the decision function:

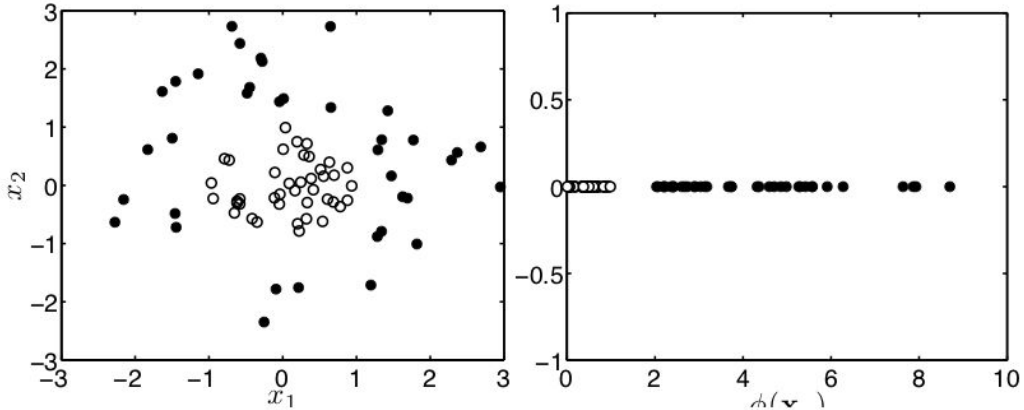
$$t_{\text{new}} = \text{sign}\left(\sum_n \alpha_n t_n \mathbf{x}_n^\mathsf{T} \mathbf{x}_{\text{new}} + b \right)$$

- Data ($\mathbf{x}_n, \mathbf{x}_m, \mathbf{x}_{\text{new}}$, etc) only appears as inner (dot) products:

$$\mathbf{x}_n^\mathsf{T} \mathbf{x}_m, \;\; \mathbf{x}_n^\mathsf{T} \mathbf{x}_{\text{new}}, \text{etc}$$

Kernel

- Our SVM can find linear decision boundaries.
- What if the data requires something nonlinear?



- We can transform the data e.g.:

$$\phi(\mathbf{x}_n) = x_{n1}^2 + x_{n2}^2$$

- So that it can be separated with a straight line.
- And use $\phi(\mathbf{x}_n)$ instead of $\mathbf{x}_n$ in our optimisation.

Kernel

- Our optimisation is now:

$$\underset{\boldsymbol{\alpha}}{\text{argmax}} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m} \alpha_n \alpha_m t_n t_m \phi(\mathbf{x}_n)^\mathsf{T} \phi(\mathbf{x}_m)$$

- And predictions:

$$t_{\text{new}} = \text{sign}\left(\sum_n \alpha_n t_n \phi(\mathbf{x}_n)^\mathsf{T} \phi(\mathbf{x}_{\text{new}}) + b \right)$$

- In this case:

$$\phi(\mathbf{x}_n^\mathsf{T})\phi(\mathbf{x}_m) = (x_{n1}^2 + x_{n2}^2)(x_{m1}^2 + x_{m2}^2) = k(\mathbf{x}_n, \mathbf{x}_m)$$

- We can think of the dot product in the projected space as a function of the original data.

Kernel

- We needn't directly think of projections at all.
- Can just think of functions $k(\mathbf{x}_n, \mathbf{x}_m)$ that *are dot products in some space*.
- Called *kernel* functions.
- Don't ever need to actually project the data – just use the kernel function to compute what the dot product would be if we did project.
- Optimisation task:

$$\underset{\boldsymbol{\alpha}}{\text{argmax}} \sum_n \alpha_n - \frac{1}{2} \sum_{n,m} \alpha_n \alpha_m t_n t_m k(\mathbf{x}_n, \mathbf{x}_m)$$

- Predictions:

$$t_{\text{new}} = \text{sign}\left(\sum_n \alpha_n t_n k(\mathbf{x}_n, \mathbf{x}_{\text{new}}) + b \right)$$

Kernel

- Plenty of off-the-shelf kernels that we can use:
- Linear:

$$k(\mathbf{x}_n, \mathbf{x}_m) = \mathbf{x}_n^{\mathsf{T}} \mathbf{x}_m$$

- Gaussian:

$$k(\mathbf{x}_n, \mathbf{x}_m) = \exp\left\{-\beta(\mathbf{x}_n - \mathbf{x}_m)^{\mathsf{T}}(\mathbf{x}_n - \mathbf{x}_m)\right\}$$

- Polynomial:

$$k(\mathbf{x}_n, \mathbf{x}_m) = (1 + \mathbf{x}_n^{\mathsf{T}} \mathbf{x}_m)^{\beta}$$

- These all correspond to $\phi(\mathbf{x}_n)^{\mathsf{T}} \phi(\mathbf{x}_m)$ for some transformation $\phi(\mathbf{x}_n)$.
- Don't know what the projections $\phi(\mathbf{x}_n)$ are – don't need to know!

Kernel

- Our algorithm is still only finding linear boundaries....
- ...but we're finding linear boundaries in some other space.
- The optimisation is just as simple, regardless of the kernel choice.
  - Still a quadratic program.
  - Still a single, global optimum.
- We can find very complex decision boundaries with a linear algorithm!

A technical point

- Our decision boundary was defined as $\mathbf{w}^\mathsf{T}\mathbf{x} + b = 0$.
- Now, $\mathbf{w}$ is defined as:

$$\mathbf{w} = \sum_{n=1}^{N} \alpha_n t_n \phi(\mathbf{x}_n)$$

- We don't know $\phi(\mathbf{x}_n)$.
- We **only know** $\phi(\mathbf{x}_n)^\mathsf{T}\phi(\mathbf{x}_m) = k(\mathbf{x}_n, \mathbf{x}_m)$
- So, we can't compute $\mathbf{w}$ or the boundary!
- But we can evaluate the predictions on a grid of $\mathbf{x}_{\text{new}}$ and use Matlab to draw a contour:

$$\sum_{n=1}^{N} \alpha_n t_n k(\mathbf{x}_n, \mathbf{x}_{\text{new}}) + b$$

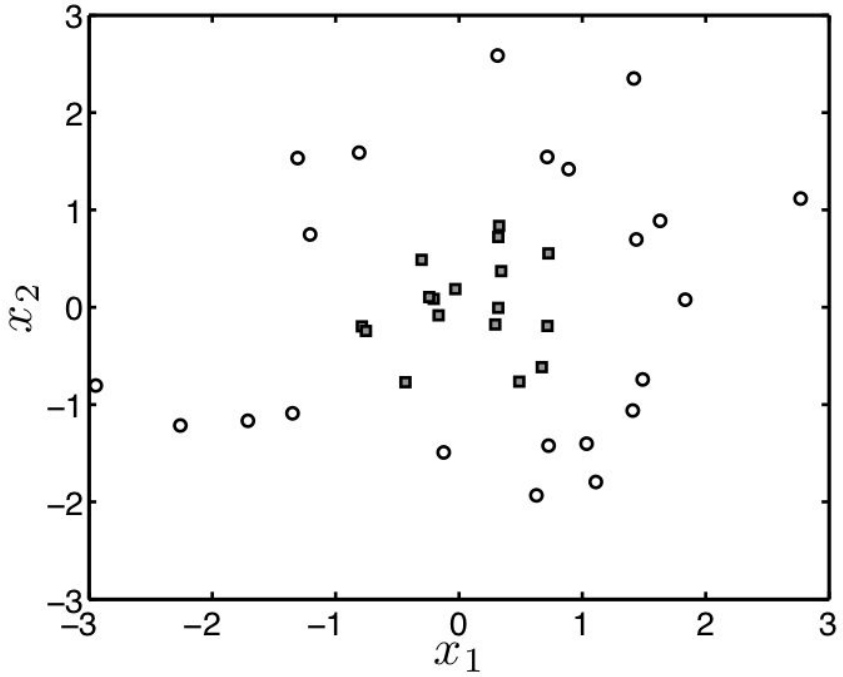
Aside: kernelising other algorithms

- **Many** algorithms can be kernelised.
  - Any that can be written with data only appearing as inner products.
- Simple algorithms can be used to solve very complex problems!
- Class exercise:
  - KNN requires the distance between $\mathbf{x}_{\text{new}}$ and each $\mathbf{x}_n$:

$$(\mathbf{x}_{\text{new}} - \mathbf{x}_n)^\top (\mathbf{x}_{\text{new}} - \mathbf{x}_n)$$

  - Can we kernelise it?

Example: nonlinear data

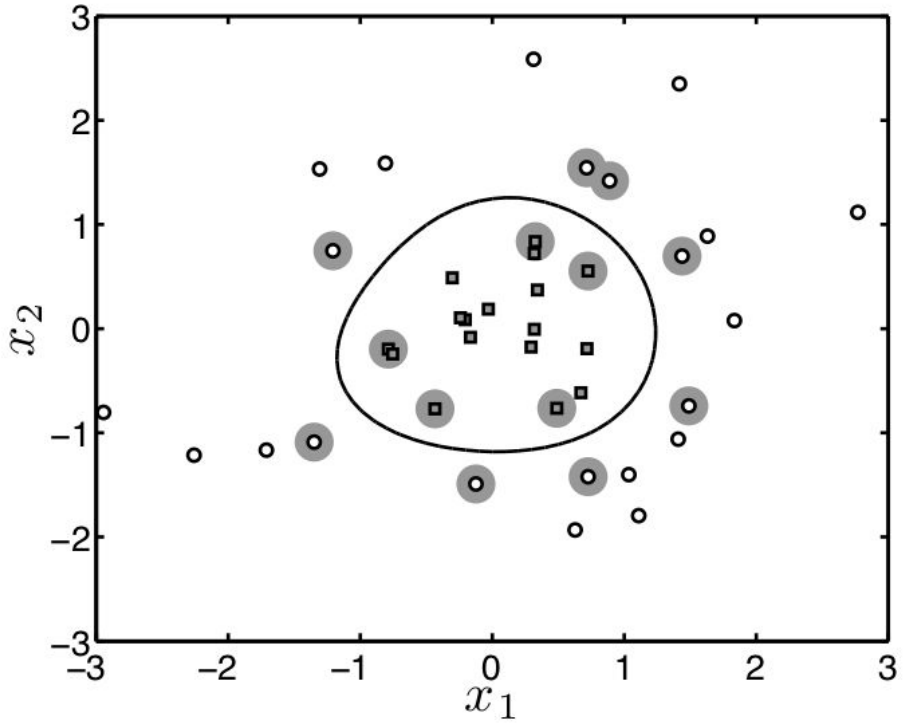


- We'll use a Gaussian kernel:

$$k(\mathbf{x}_n, \mathbf{x}_m) = \exp\left\{-\beta(\mathbf{x}_n - \mathbf{x}_m)^{\mathsf{T}}(\mathbf{x}_n - \mathbf{x}_m)\right\}$$

- And vary β ($C = 10$).

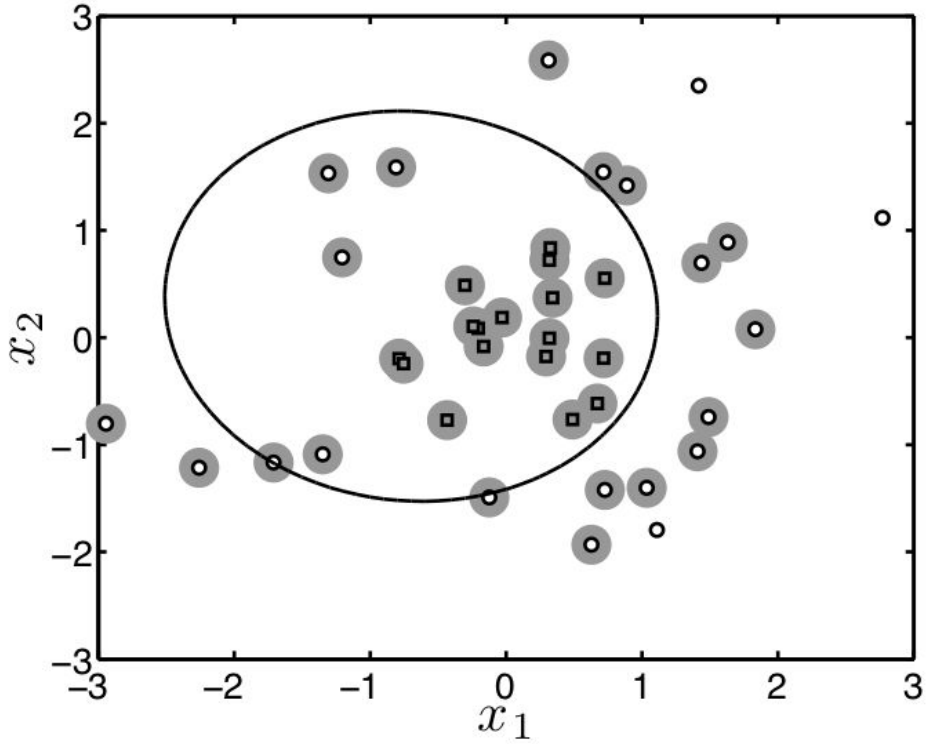
Example



- $\beta = 1.$

$$k(\mathbf{x}_n, \mathbf{x}_m) = \exp\left\{-\beta(\mathbf{x}_n - \mathbf{x}_m)^{\mathsf{T}}(\mathbf{x}_n - \mathbf{x}_m)\right\}$$

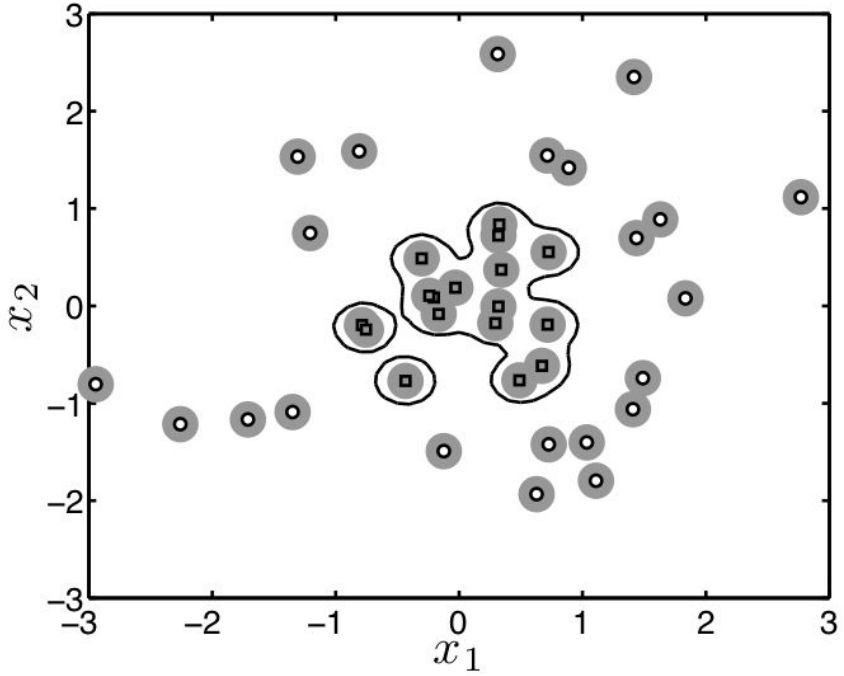
Example



- $\beta = 0.01$.

$$k(\mathbf{x}_n, \mathbf{x}_m) = \exp\left\{-\beta(\mathbf{x}_n - \mathbf{x}_m)^{\mathsf{T}}(\mathbf{x}_n - \mathbf{x}_m)\right\}$$

Example



- $\beta = 50$.

$$k(\mathbf{x}_n, \mathbf{x}_m) = \exp\left\{-\beta(\mathbf{x}_n - \mathbf{x}_m)^{\mathsf{T}}(\mathbf{x}_n - \mathbf{x}_m)\right\}$$

The Gaussian kernel

- β controls the *complexity* of the decision boundaries.
- $\beta = 0.01$ was too simple:
  - Not flexible enough to surround just the square class.
- $\beta = 50$ was too complex:
  - *Memorises* the data.
- $\beta = 1$ was about right.
- Neither $\beta = 50$ or $\beta = 0.01$ will *generalise* well.
- Both are also non-sparse (lots of support vectors).

Choosing kernel function, parameters and C

- Kernel function and parameter choice is data dependent.
- Easy to overfit.
- Need to set C too
- C and β are *linked*
  - C too high – overfitting.
  - C too low – underfitting.
- Cross-validation!
- Search over β and C
  - SVM scales with N^3 (naive implementation)
  - For large N, cross-validation over many C and β values is infeasible.

Summary - SVMs

- Described a classifier that is optimised by maximising the *margin*.
- Did some re-arranging to turn it into a quadratic programming problem.
- Saw that data only appear as inner products.
- Introduced the idea of kernels.
- Can fit a linear boundary in some other space without explicitly projecting.
- Loosened the SVM constraints to allow points on the wrong side of boundary.
- Other algorithms can be kernelised...we'll see a clustering one in the future.

Machine Learning & Artificial Intelligence for Data Scientists: Clustering (Part 1)

Ke Yuan
https://kyuanlab.org/
School of Computing Science

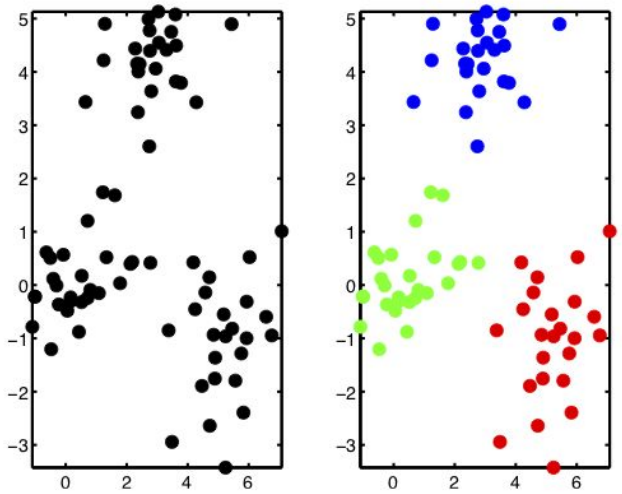
Unsupervised learning

- Everything we've seen so far has been *supervised*
- We were given a set of $\mathbf{x}_n$ **and** associated t_n.
- What if we just have $\mathbf{x}_n$?
- For example:
  - $\mathbf{x}_n$ is a binary vector indicating products customer n has bought.
  - Can group customers that buy similar products.
  - Can group products bought together.
- Known as **Clustering**
- And is an example of *unsupervised* learning.
- We'll also cover *projection* (next week)

Aims

- Understand what clustering is.
- Understand the K-means algorithm.
- Understand the idea of mixture models.

Clustering



- In this example each object has two attributes:

$$\mathbf{x}_n = [x_{n1}, x_{n2}]^{\mathsf{T}}$$

- Left: data.
- Right: data after clustering (points coloured according to cluster membership).

What we'll cover

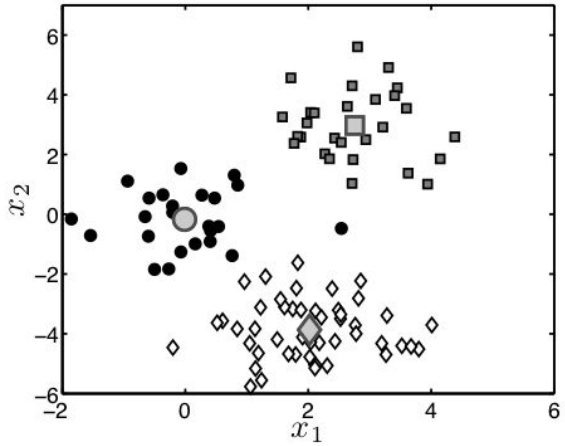
- 2 algorithms
  - K-means
  - Mixture models
- The two are related
- We'll also see how K-means can be kernelised

K-means

- Assume that there are K clusters.
- Each cluster is defined by a position in the input space:

$$\boldsymbol{\mu}_k = [\mu_{k1}, \mu_{k2}]^{\mathsf{T}}$$

- Each $\mathbf{x}_n$ is assigned to its closest cluster:



- Distance is normally Euclidean distance:

$$d_{nk} = (\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathsf{T}}(\mathbf{x}_n - \boldsymbol{\mu}_k)$$

How do we find the cluster means

- No analytical solution – we can't write down $\boldsymbol{\mu}_k$ as a function of $\mathbf{X}$.
- Use an iterative algorithm:
  1. Guss $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K$
  2. Assign each $\mathbf{x}_n$ to its closest $\boldsymbol{\mu}_k$
  3. $z_{nk} = 1$ if $\mathbf{x}_n$ assigned to $\boldsymbol{\mu}_k$ (0 otherwise)
  4. Update $\boldsymbol{\mu}_k$ to average of $\mathbf{x}_n$s assigned to $\boldsymbol{\mu}_k$:

$$\boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} z_{nk}\mathbf{x}_n}{\sum_{n=1}^{N} z_{nk}}$$

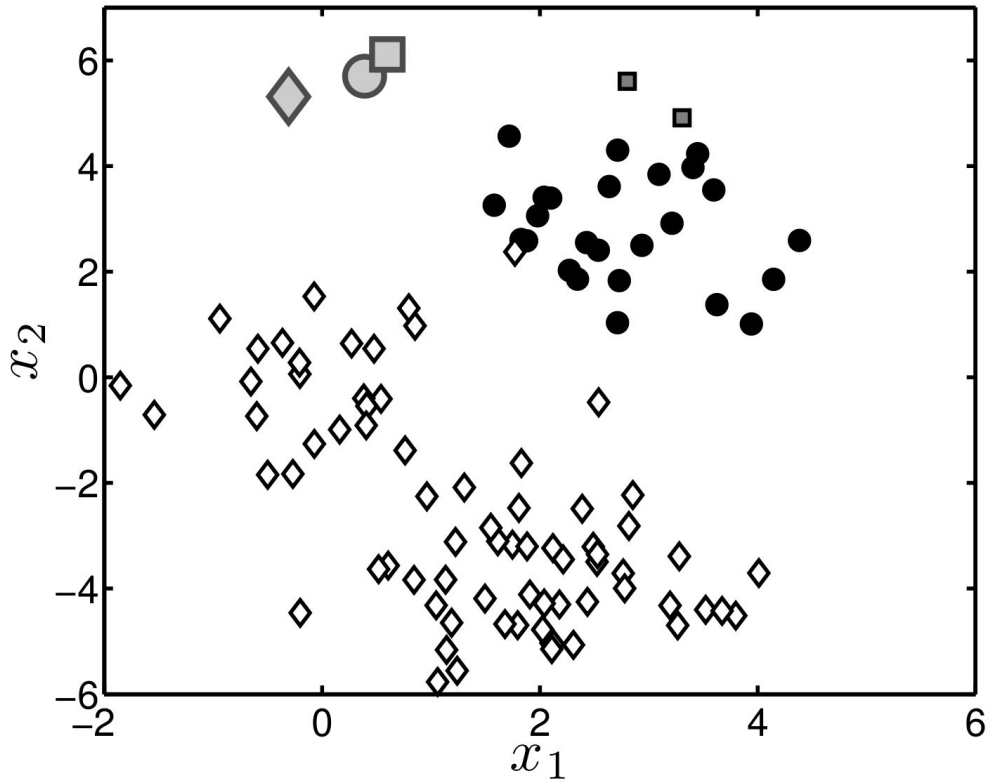
  5. Return to 2 until assignments do not change.
- Algorithm will converge....it will reach a point where the assignments don't change.

Let’s play a K-means game

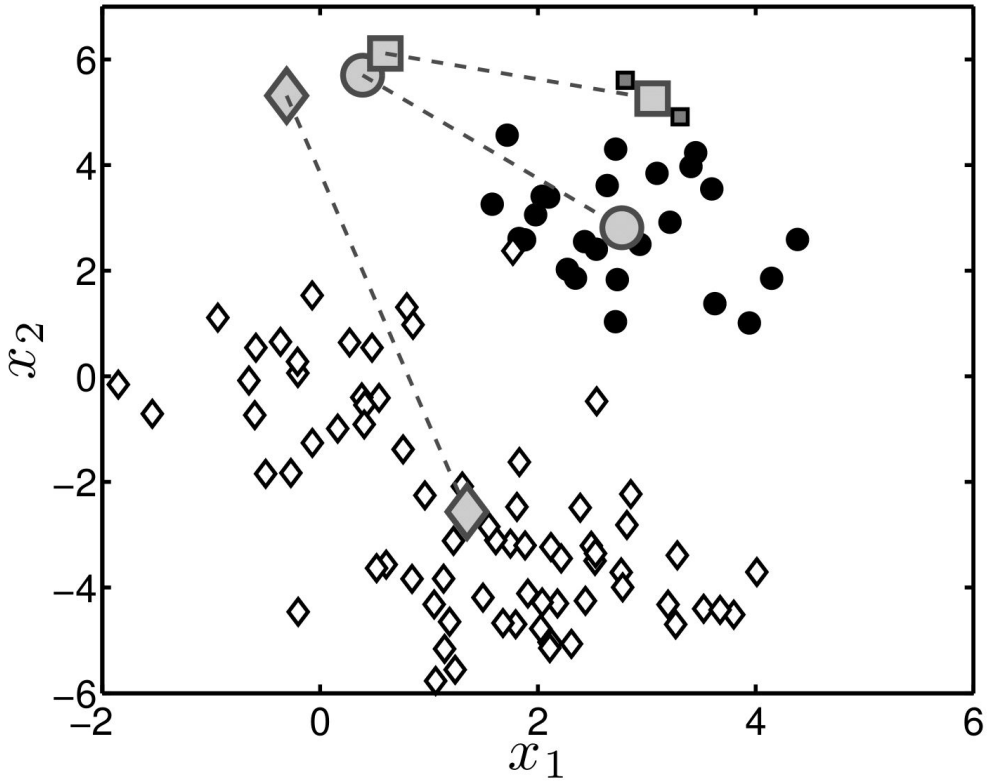
- Everyone comes to the front
- 3 volunteers act as 3 centroids
- All the rest as data points

Gather space: https://gather.town/i/Qw7GtyJN

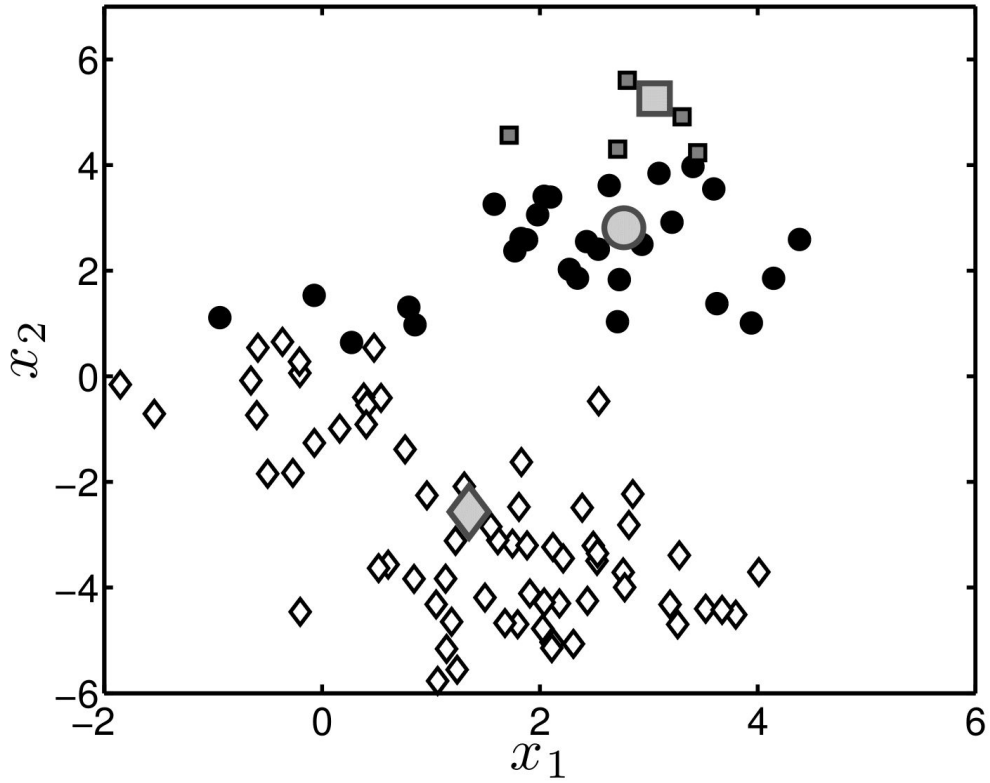
K-means - example



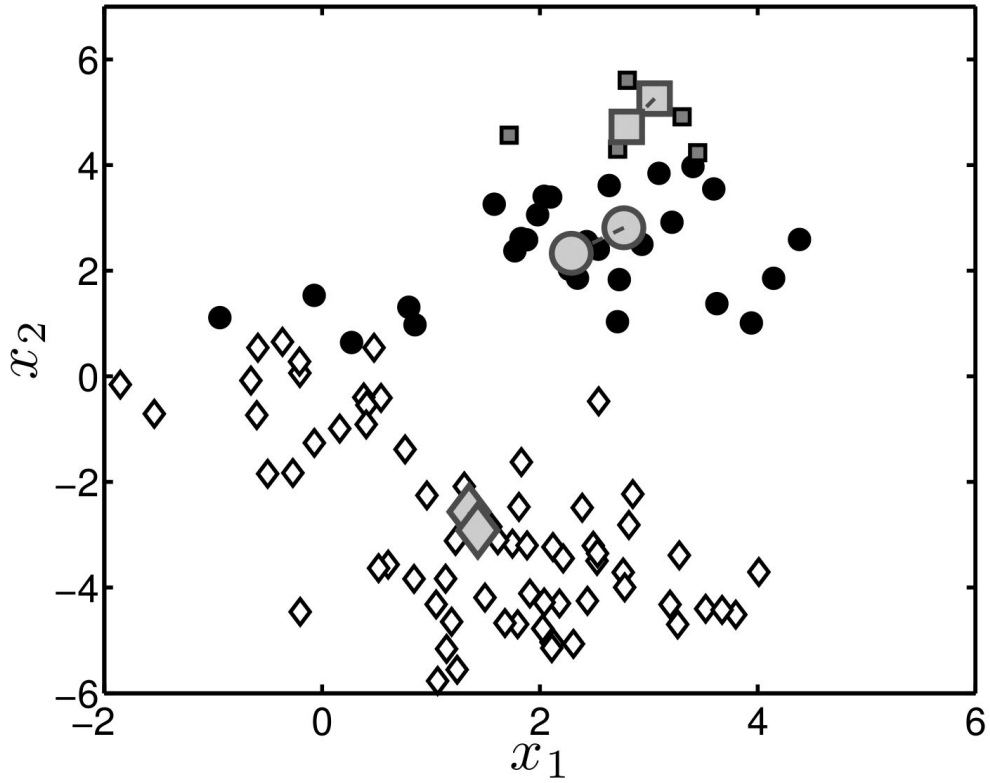
K-means - example



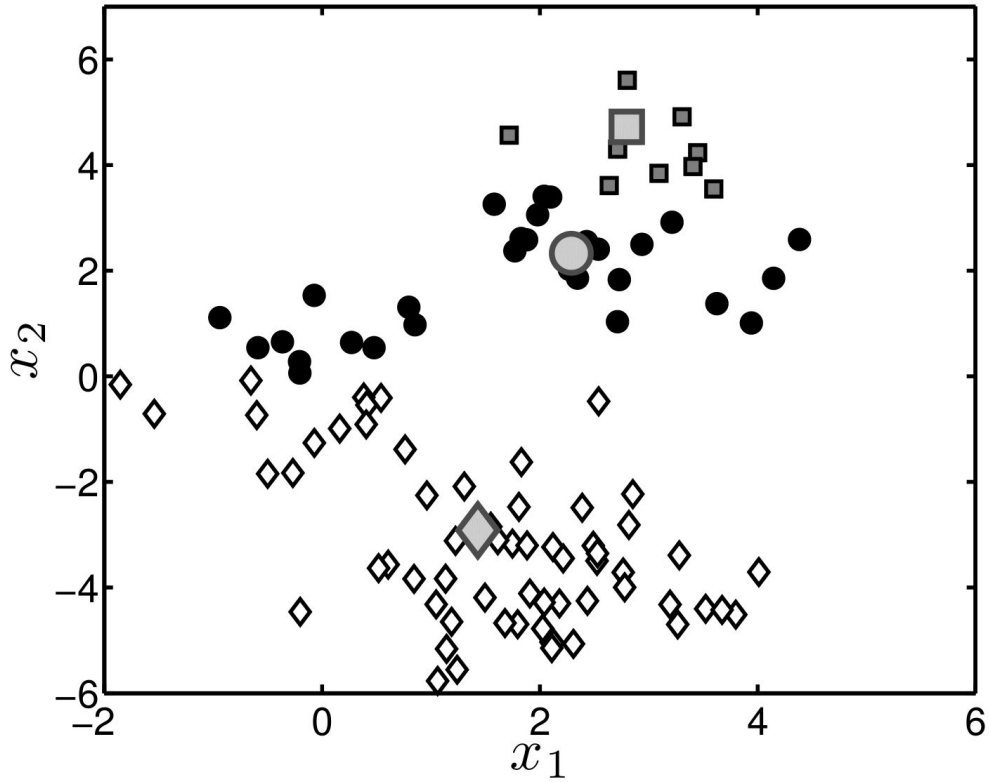
K-means - example



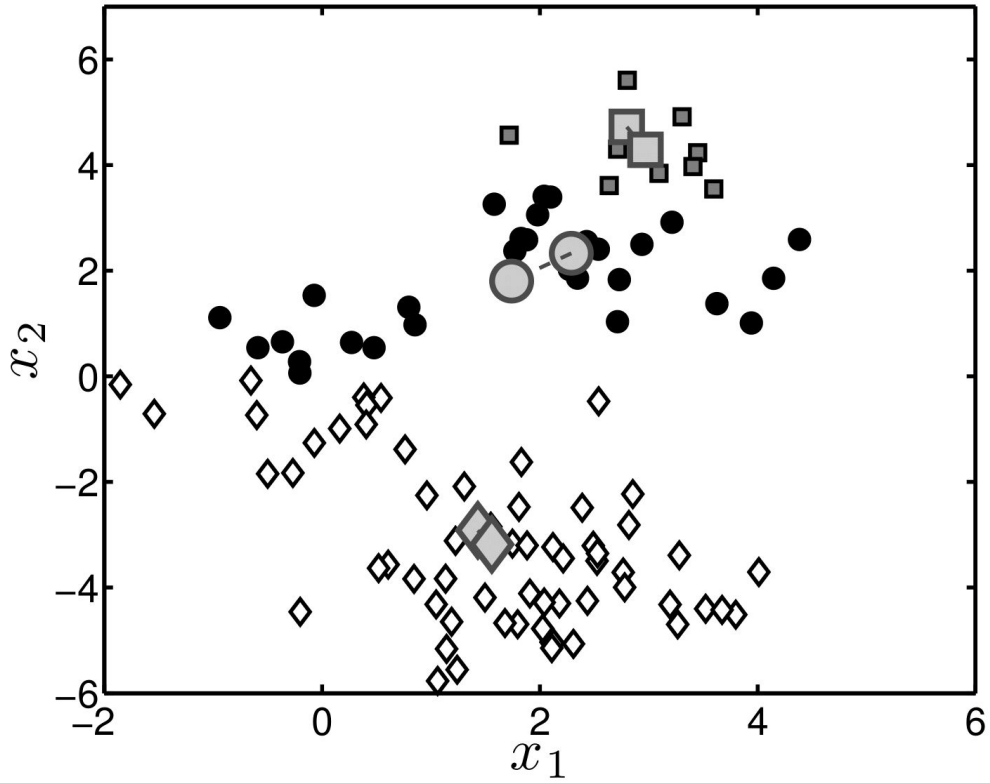
K-means - example



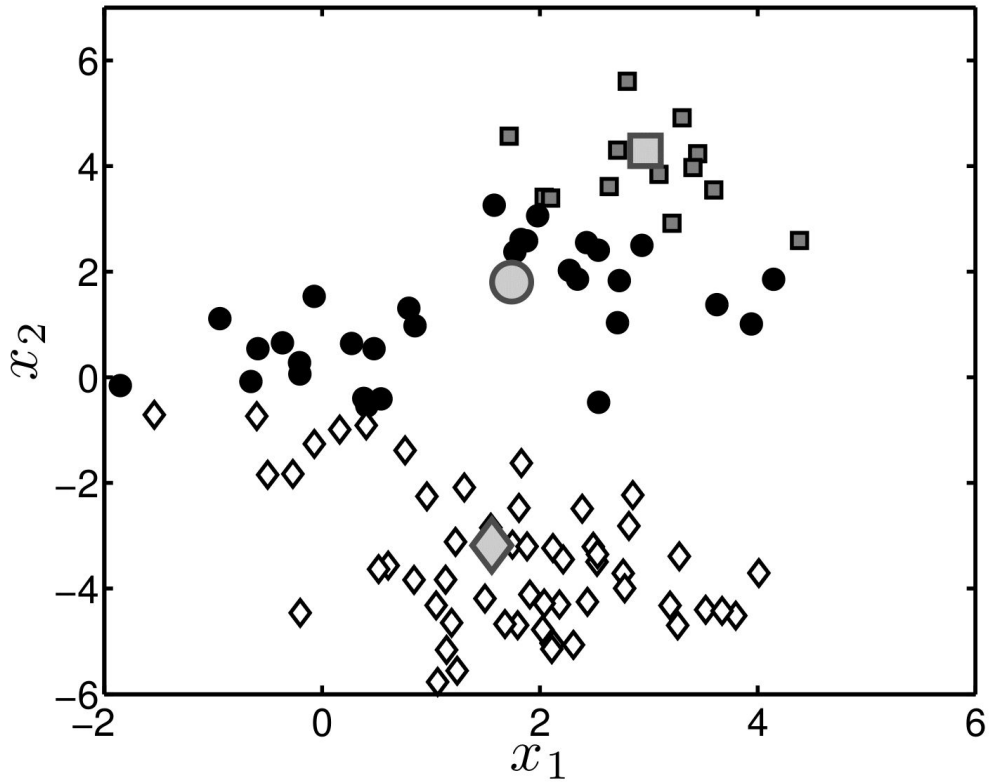
K-means - example



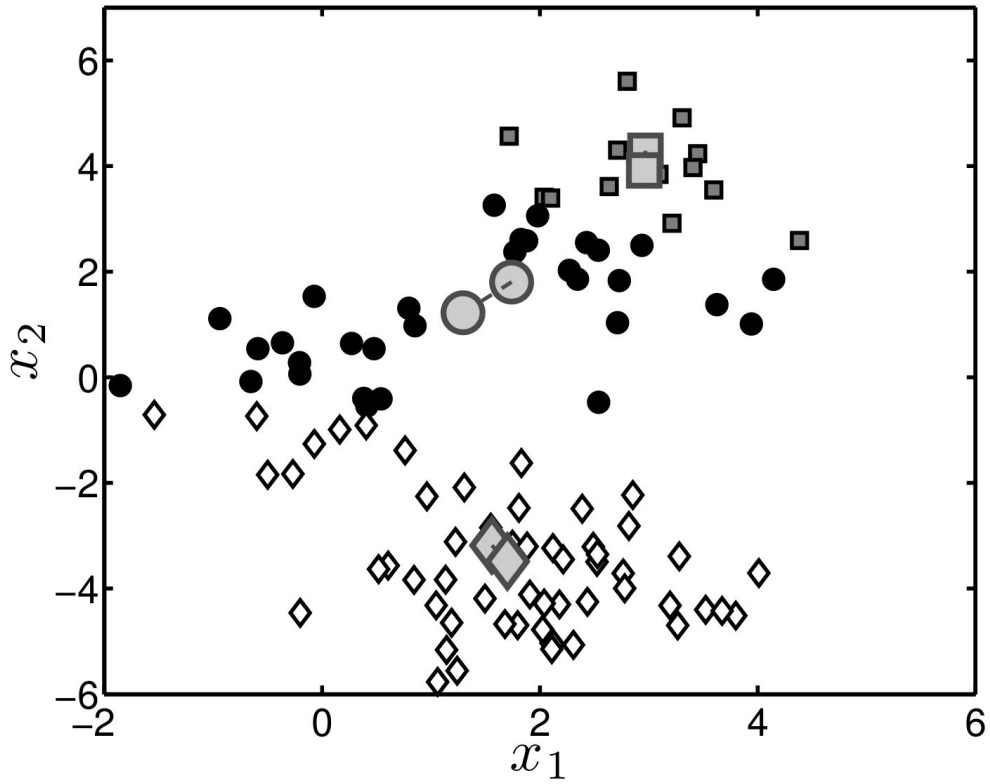
K-means - example



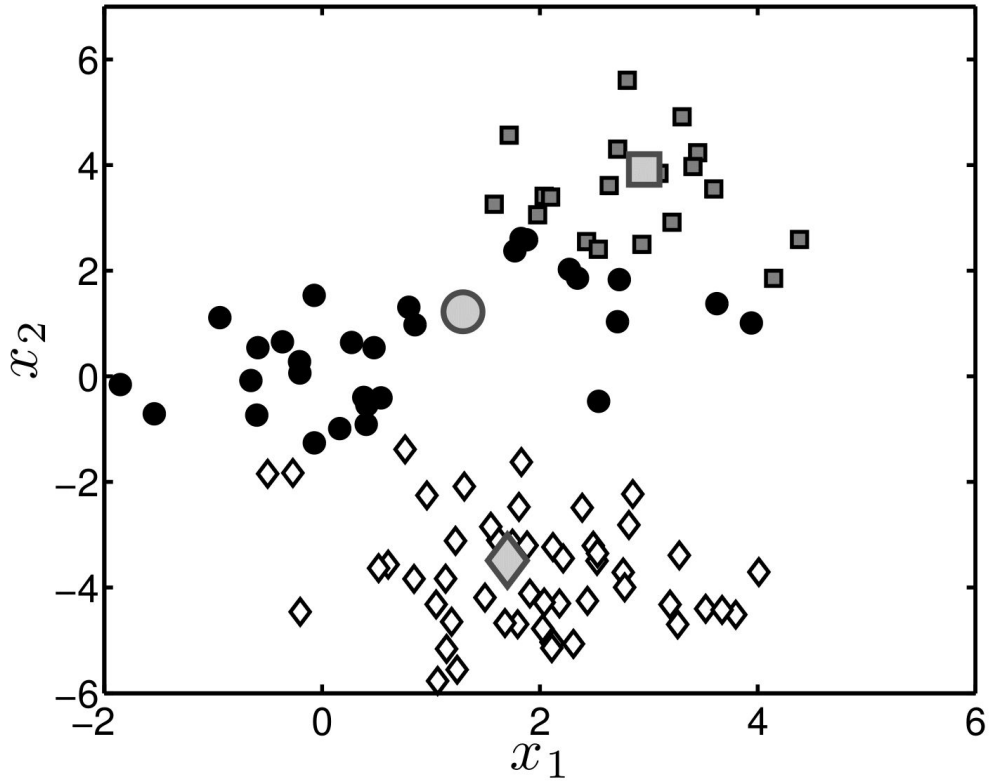
K-means - example



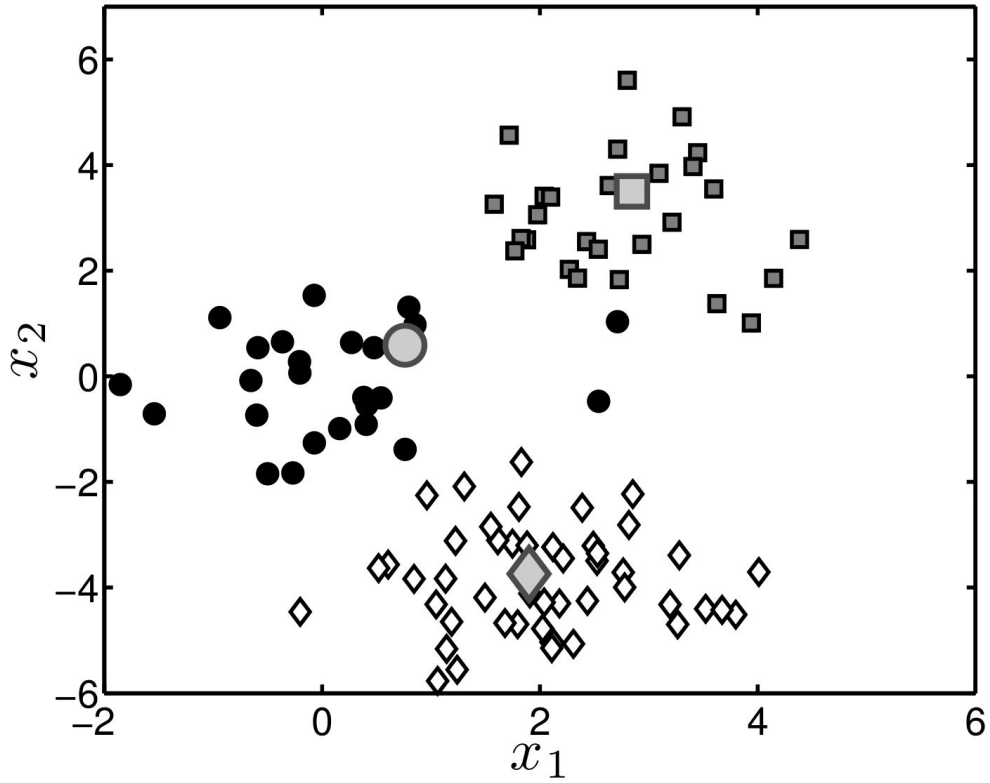
K-means - example



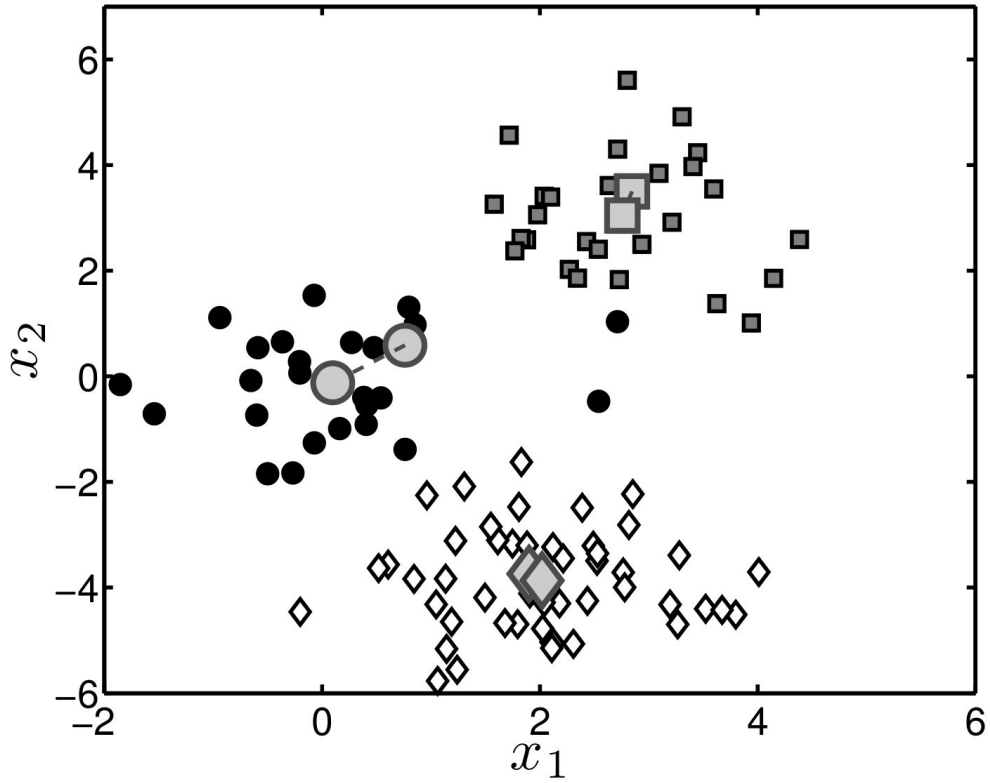
K-means - example



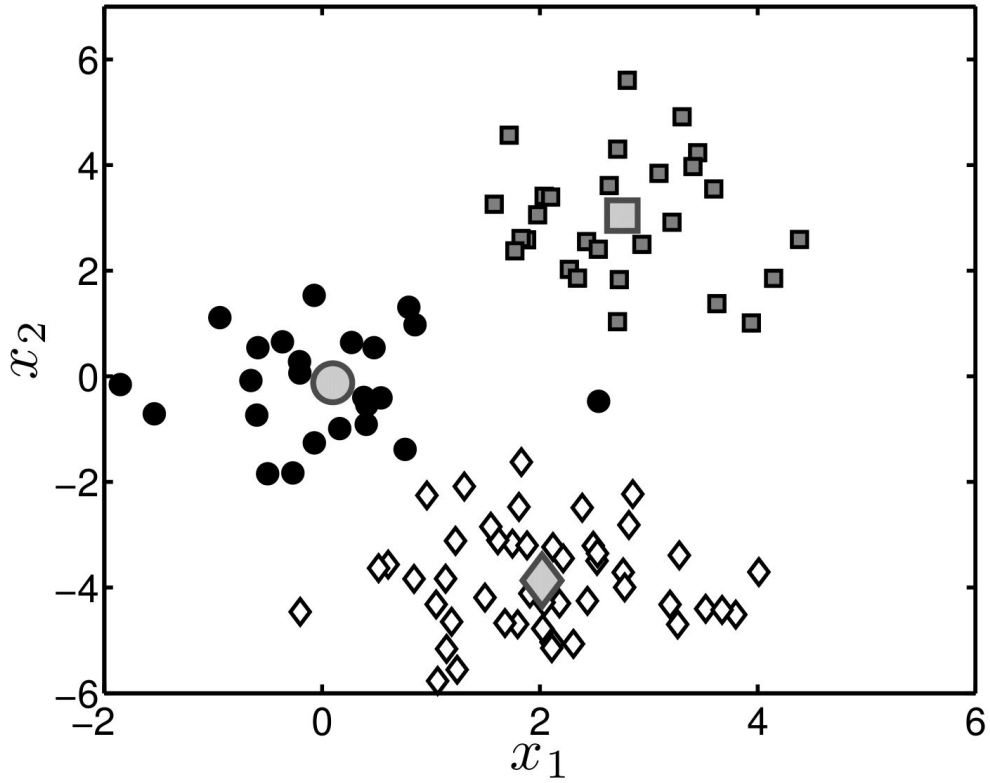
K-means - example



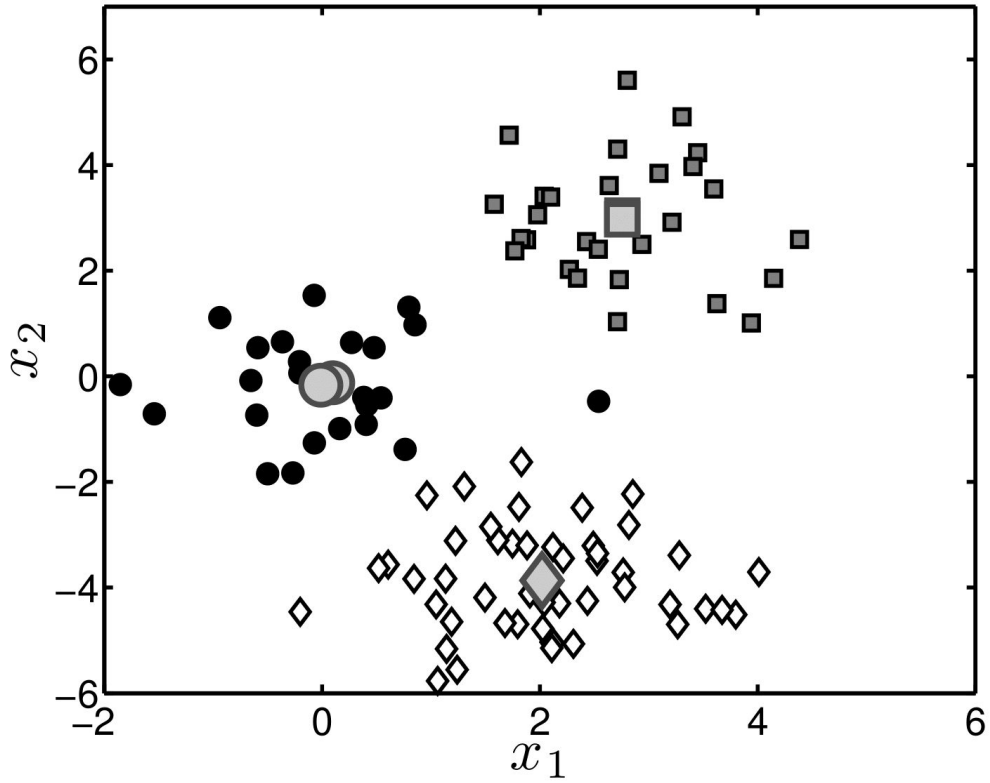
K-means - example



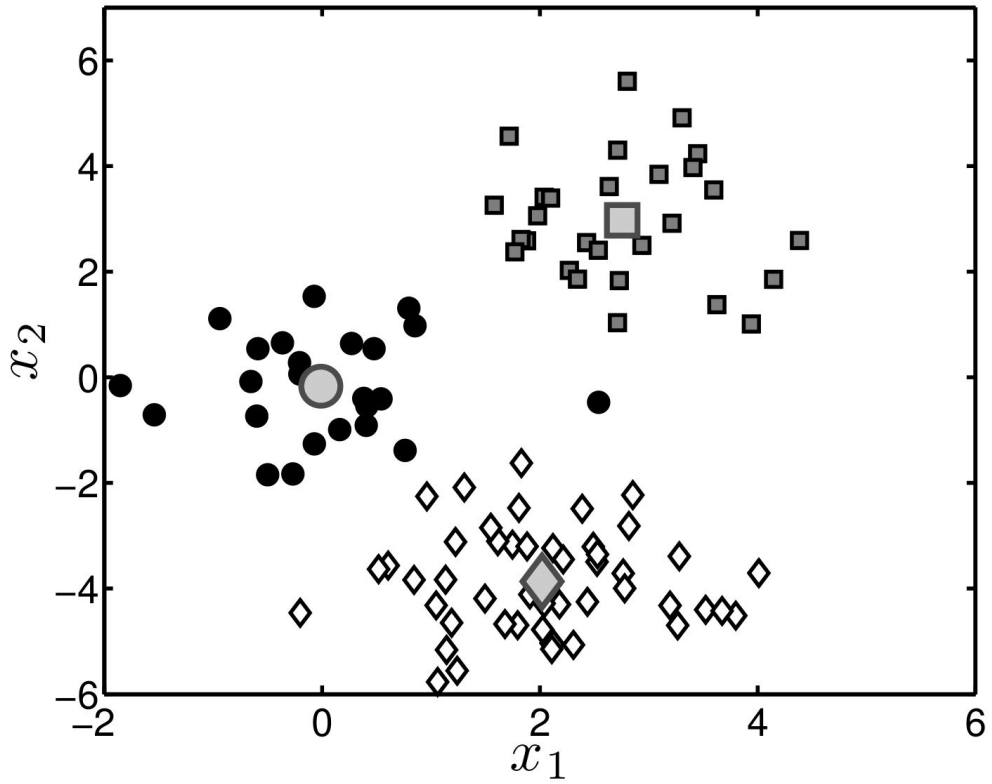
K-means - example



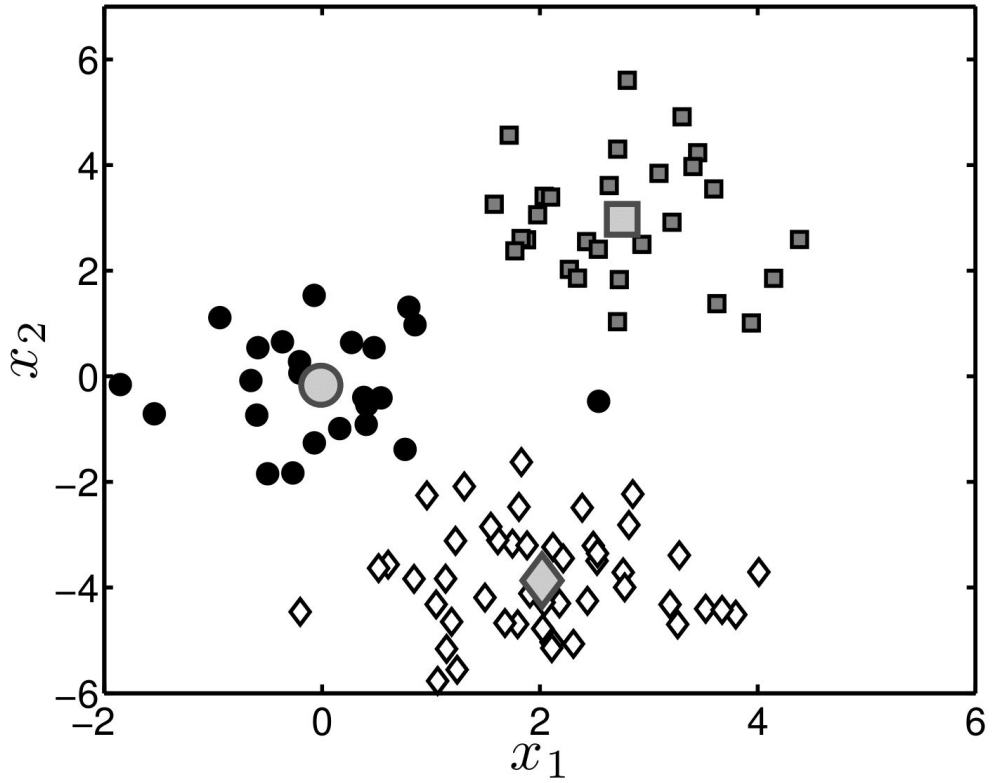
K-means - example



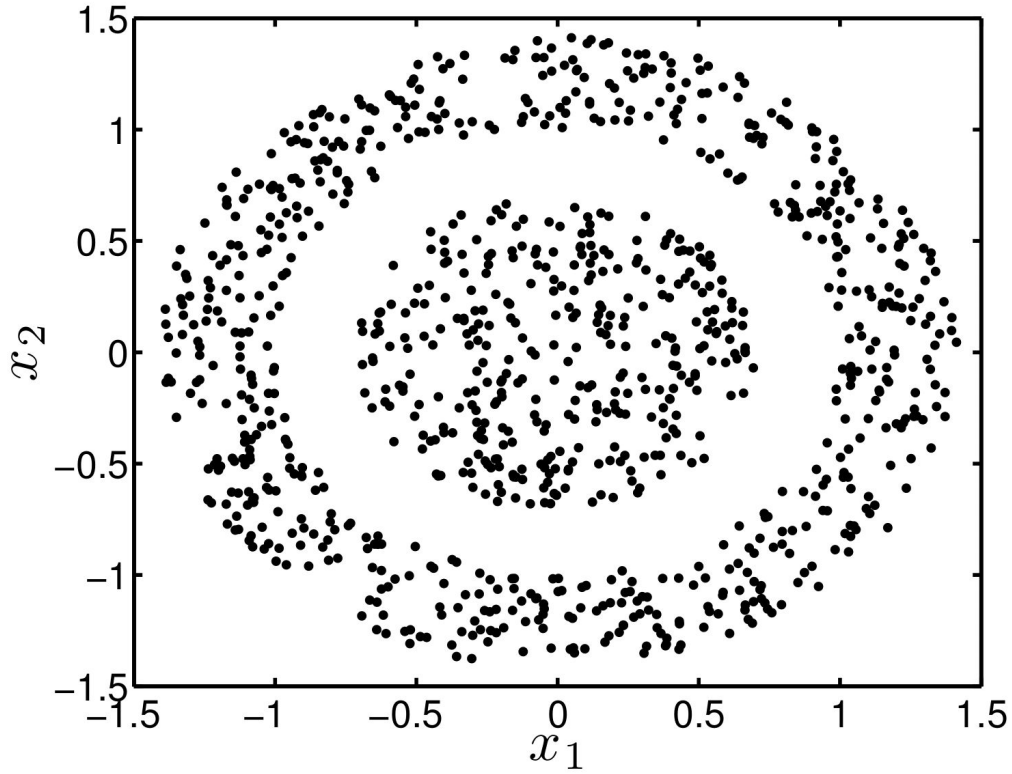
K-means - example



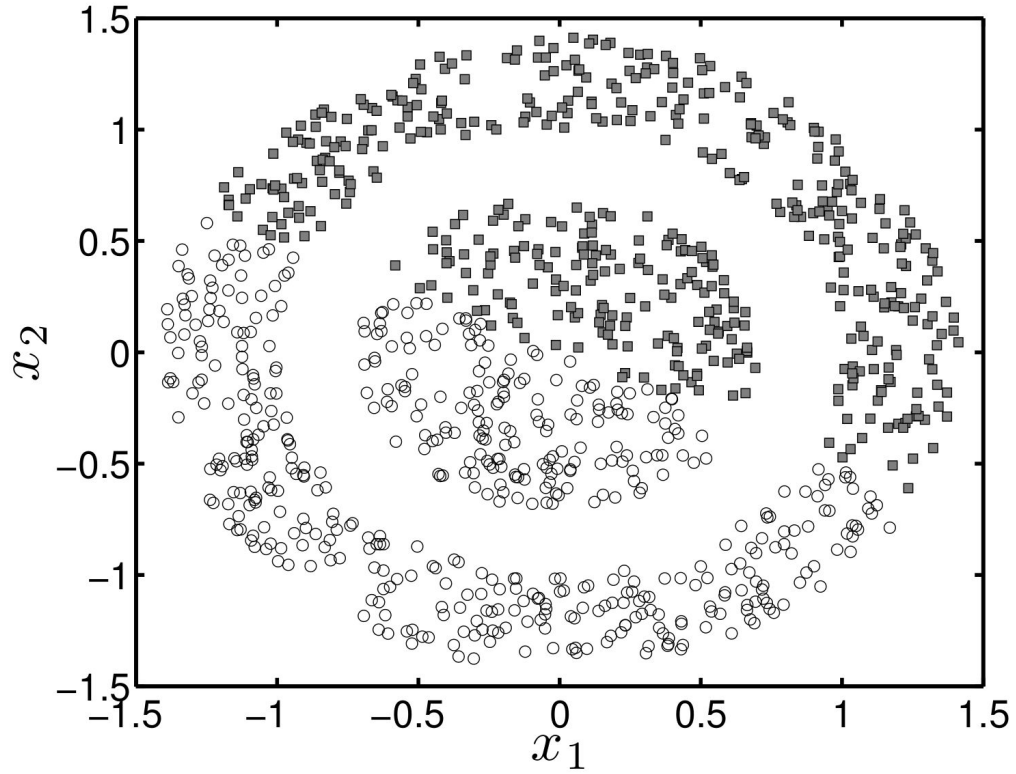
K-means - example - converged



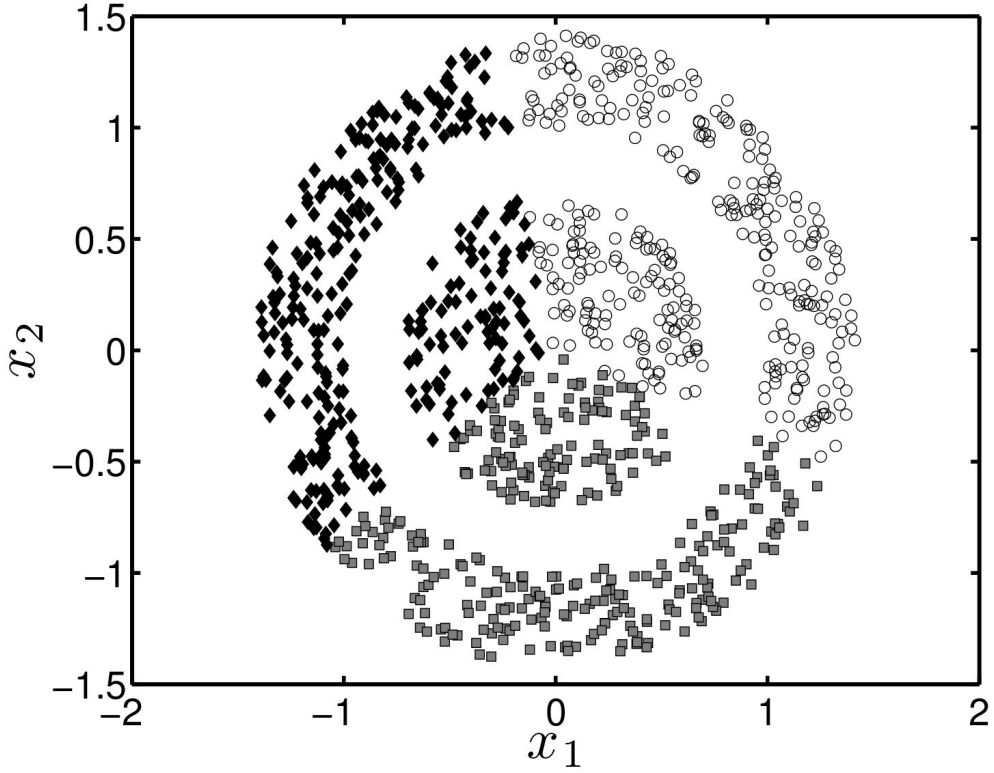
When does K-means break?



K-means with 2 clusters



K-means with 3 clusters



Kernelising K-means

$$d_{nk} = (\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathsf{T}}(\mathbf{x}_n - \boldsymbol{\mu}_k) \qquad \boldsymbol{\mu}_k = \frac{\sum_{n=1}^{N} z_{nk}\mathbf{x}_n}{\sum_{n=1}^{N} z_{nk}}$$

- Multiply out:

$$\mathbf{x}_n^{\mathsf{T}}\mathbf{x}_n - 2N_k^{-1}\sum_{m=1}^{N} z_{mk}\mathbf{x}_m^{\mathsf{T}}\mathbf{x}_n + N_k^{-2}\sum_{m,l} z_{mk}z_{lk}\mathbf{x}_m^{\mathsf{T}}\mathbf{x}_l$$

- Kernel substitution:

$$k(\mathbf{x}_n,\mathbf{x}_n) - 2N_k^{-1}\sum_{m=1}^{N} z_{mk}k(\mathbf{x}_n,\mathbf{x}_m) + N_k^{-2}\sum_{m,l=1}^{N} z_{mk}z_{lk}k(\mathbf{x}_m,\mathbf{x}_l)$$

Kernel K-means

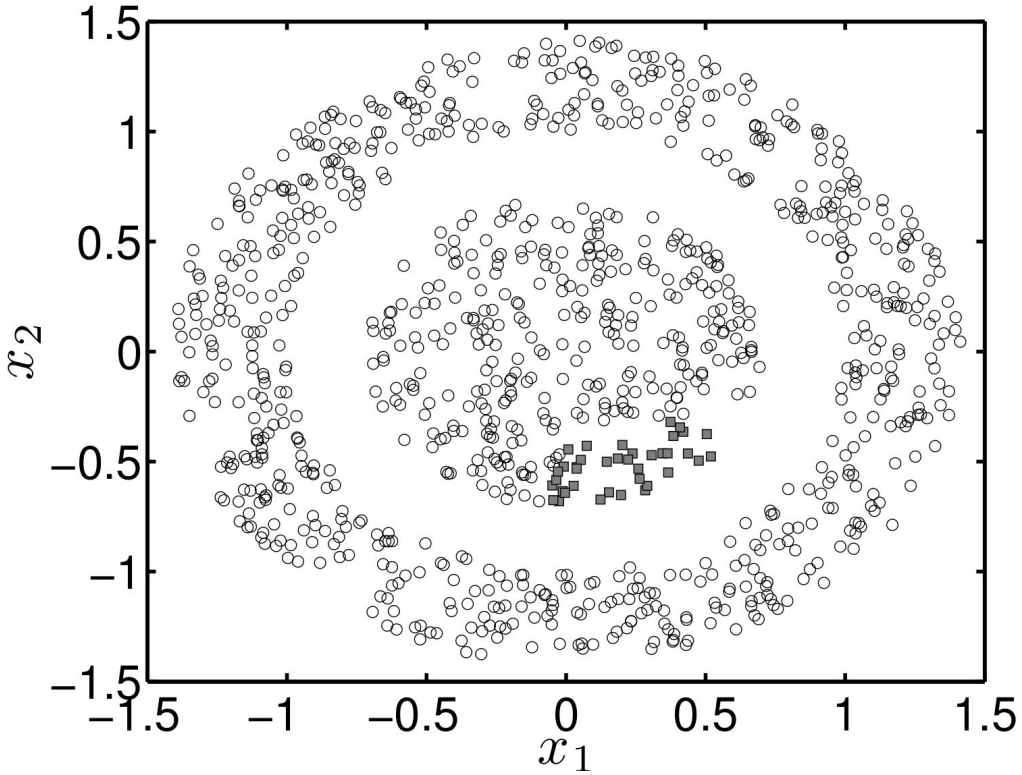
- Algorithm:
  1. Choose a kernel and any necessary parameters.
  2. Start with random assignments z_{nk}.
  3. For each $\mathbf{x}_n$ assign it to the nearest 'center' where distance is defined as:

$$k(\mathbf{x}_n, \mathbf{x}_n) - 2N_k^{-1} \sum_{m=1}^{N} z_{mk} k(\mathbf{x}_n, \mathbf{x}_m) + N_k^{-2} \sum_{m,l=1}^{N} z_{mk} z_{lk} k(\mathbf{x}_m, \mathbf{x}_l)$$

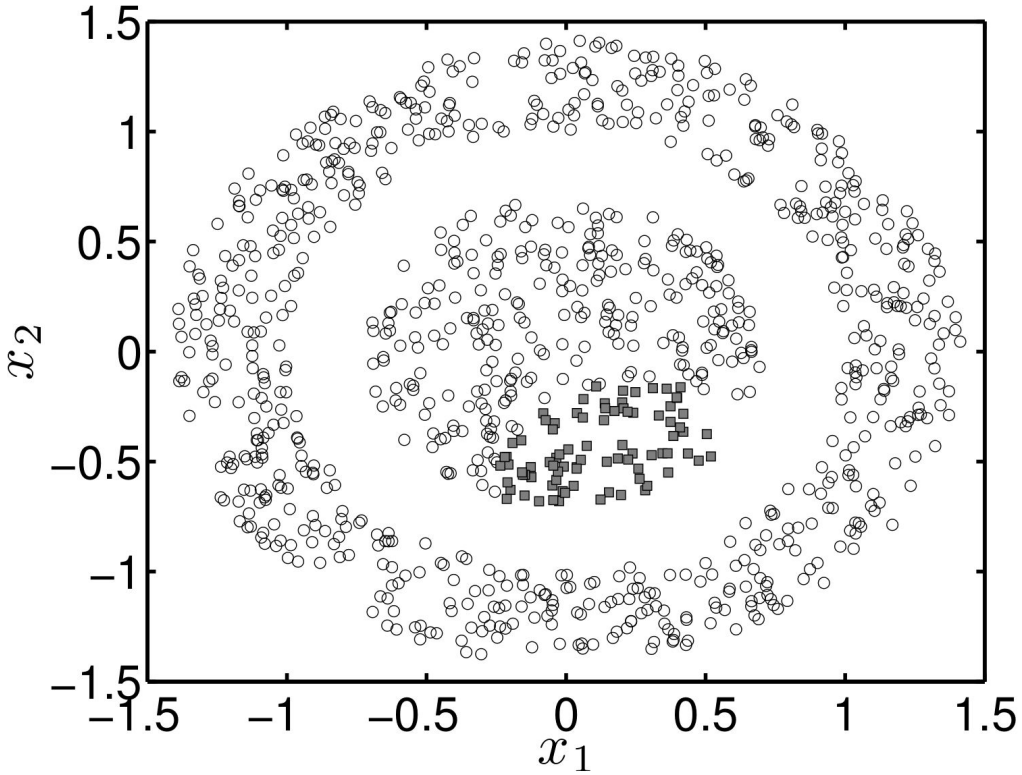
  4. If assignments have changed, return to 3.

- Note – no $\boldsymbol{\mu}_k$. This would be $N_k^{-1} \sum_n z_{nk} \phi(\mathbf{x}_n)$ but we don't know $\phi(\mathbf{x}_n)$ for kernels. We only know $\phi(\mathbf{x}_n)^{\mathsf{T}} \phi(\mathbf{x}_m)$ (last week)...

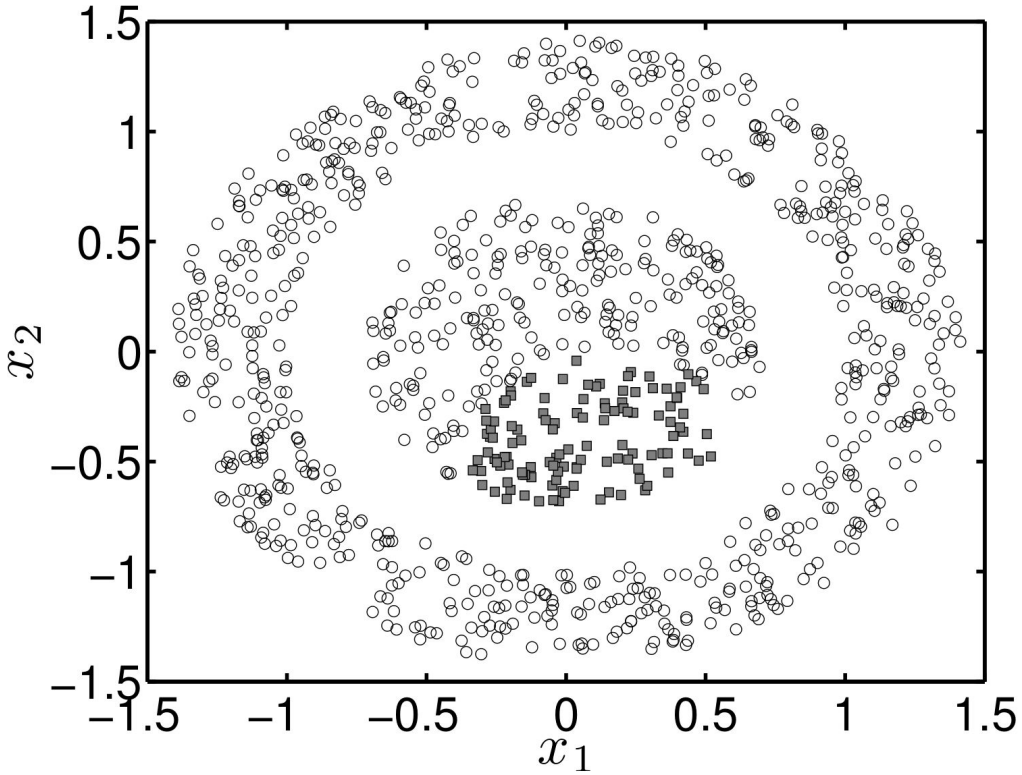
Kernel K-means - example



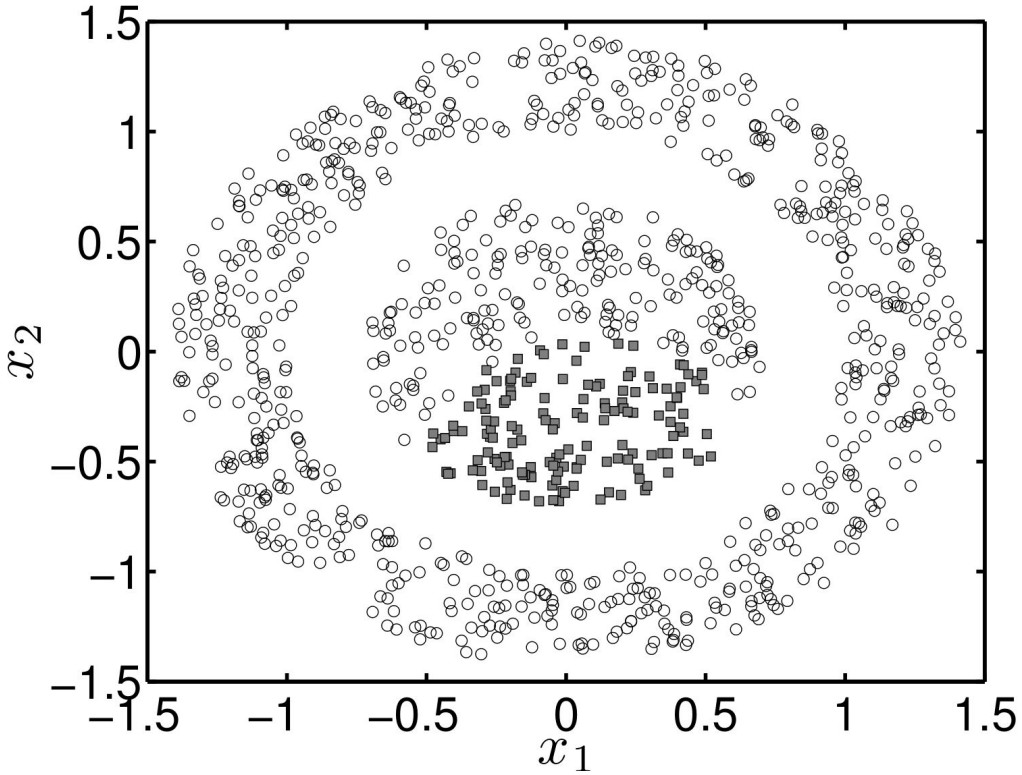
Kernel K-means - example



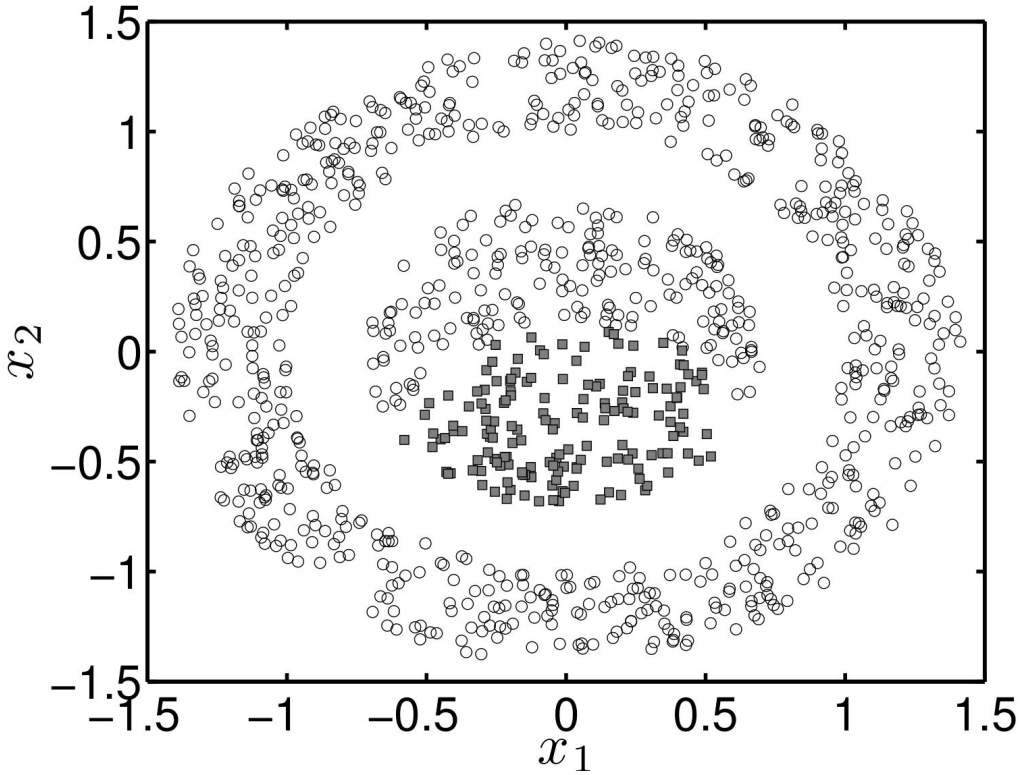
Kernel K-means - example



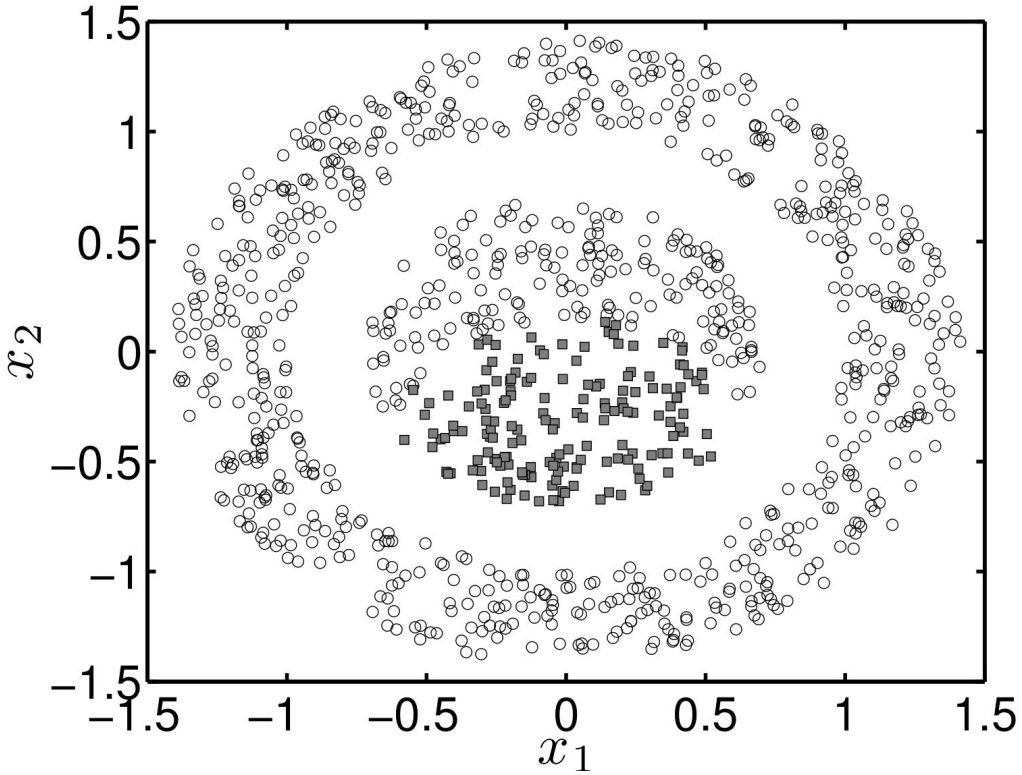
Kernel K-means - example



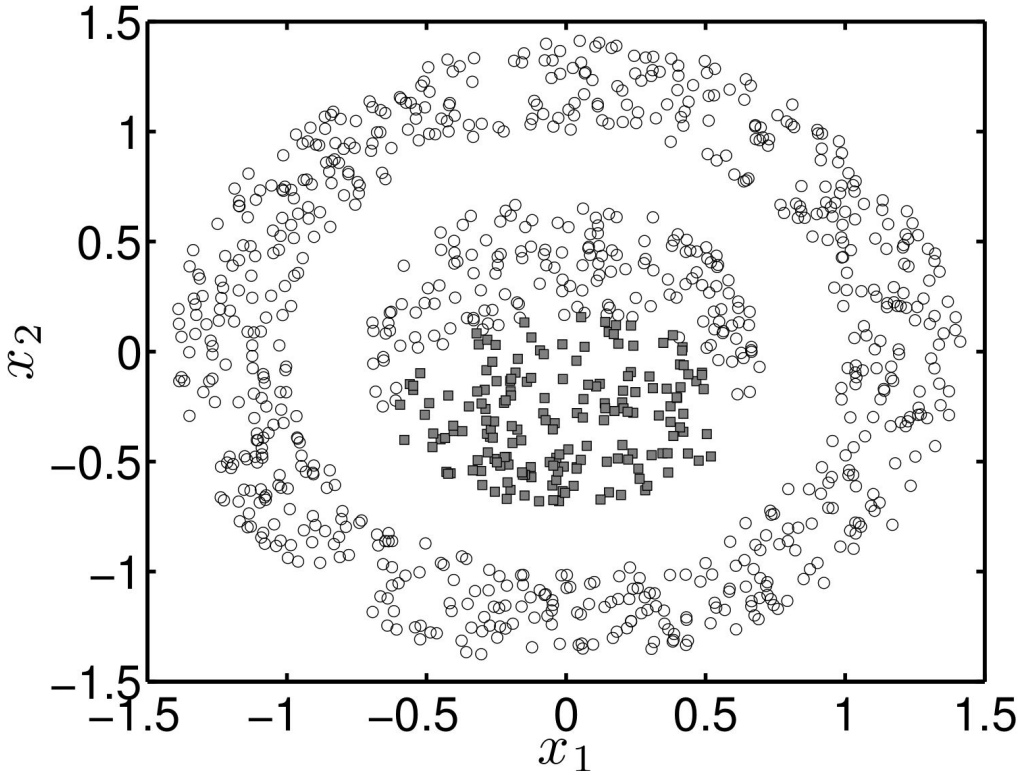
Kernel K-means - example



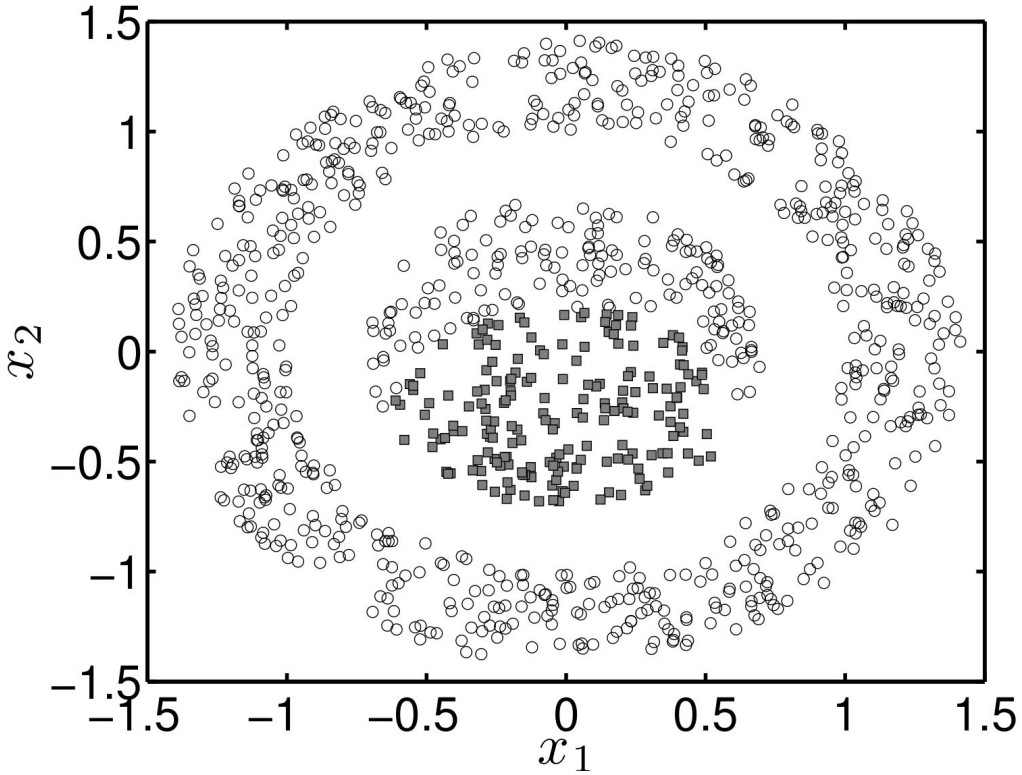
Kernel K-means - example



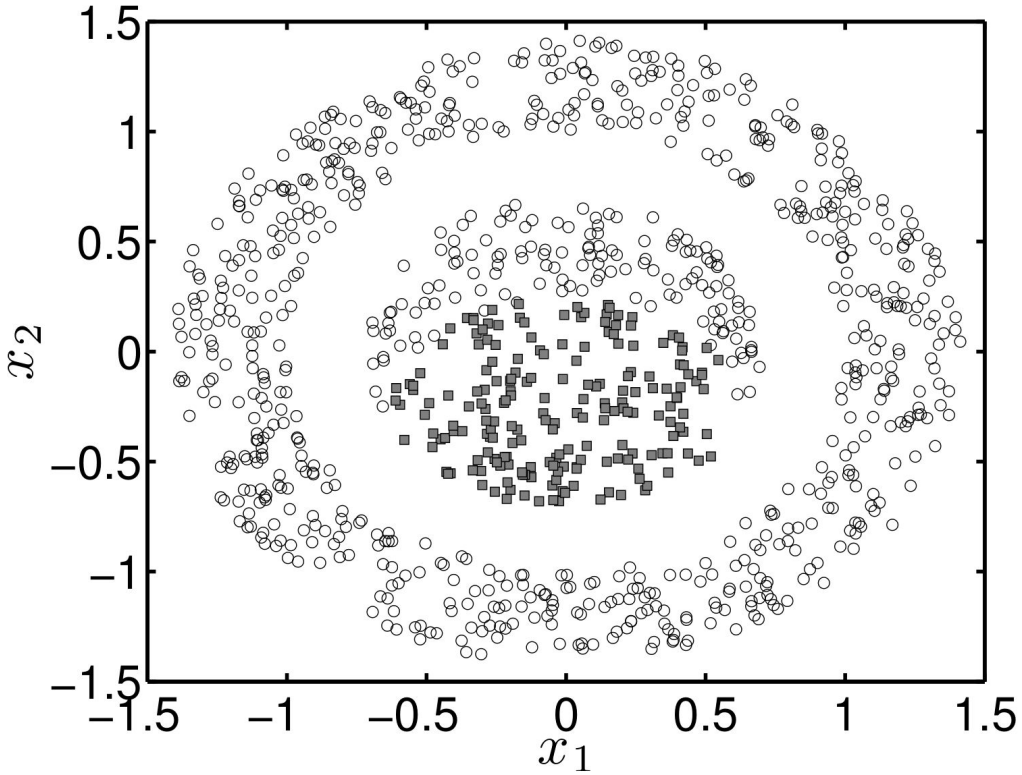
Kernel K-means - example



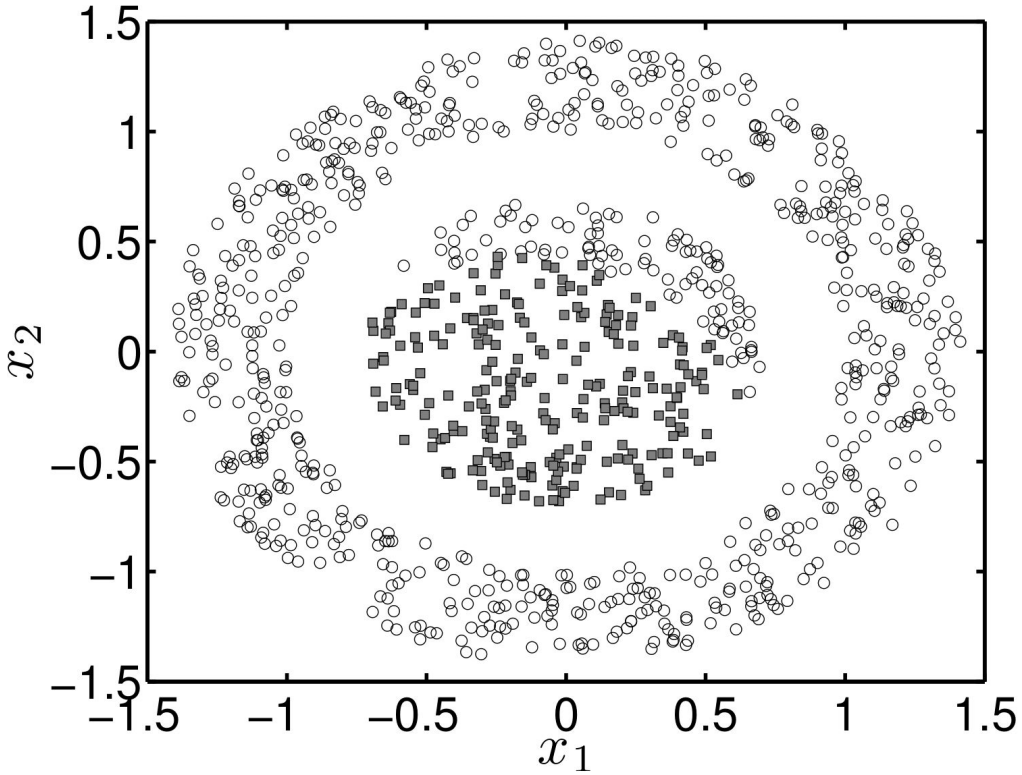
Kernel K-means - example



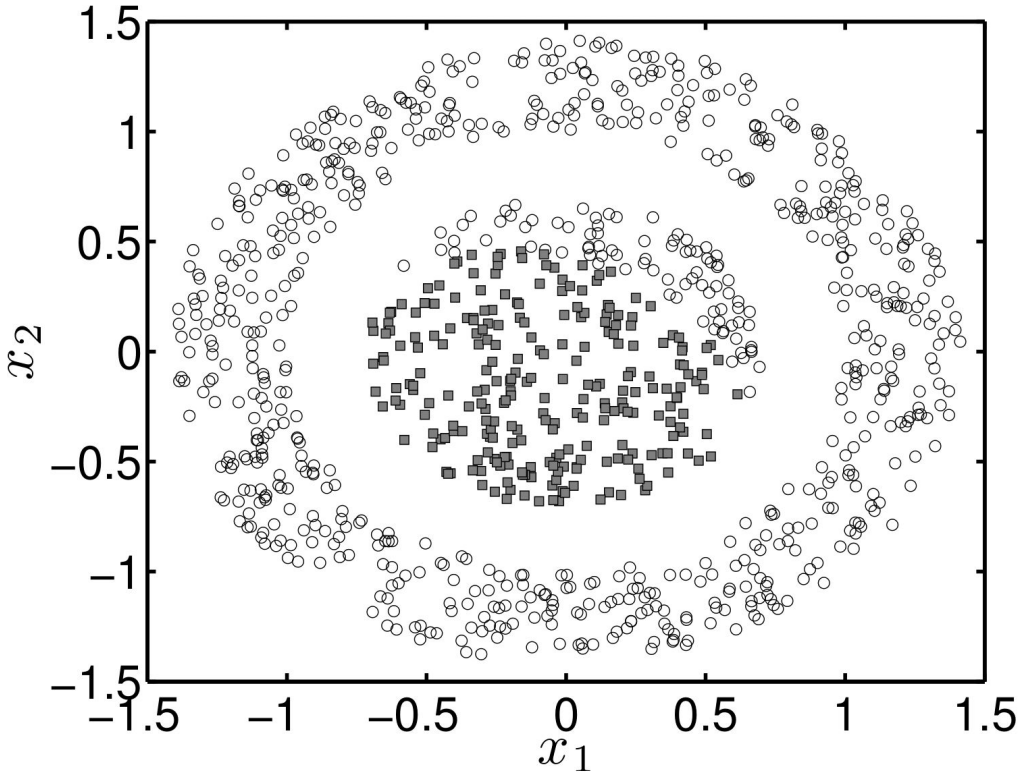
Kernel K-means - example



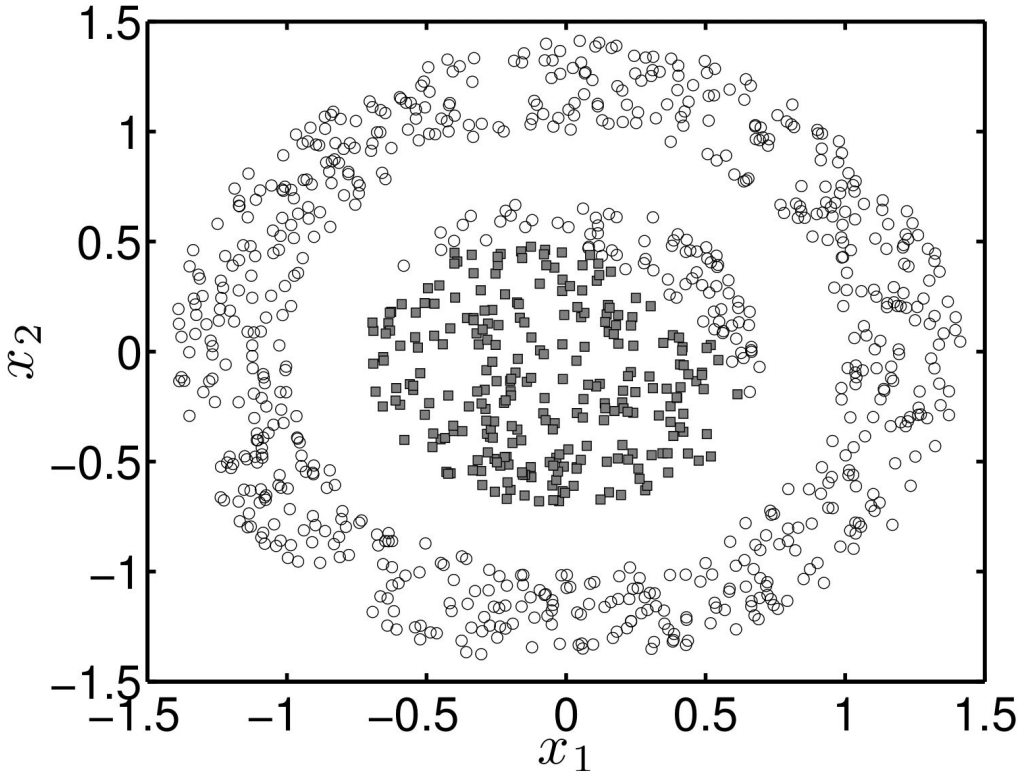
Kernel K-means - example



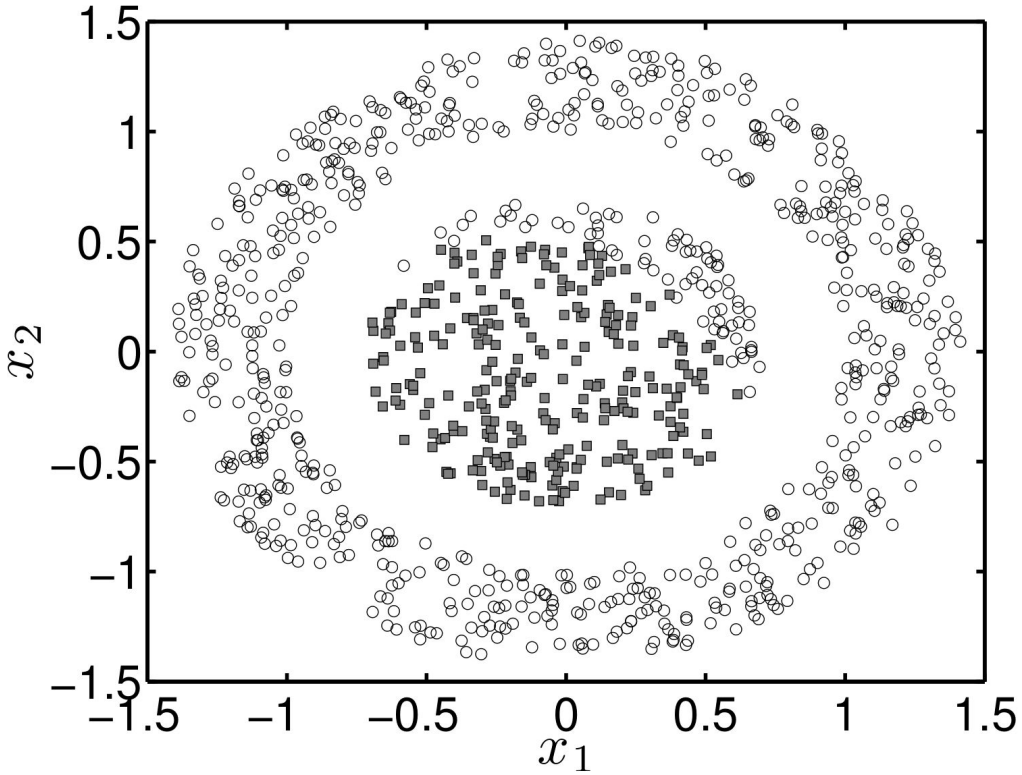
Kernel K-means - example



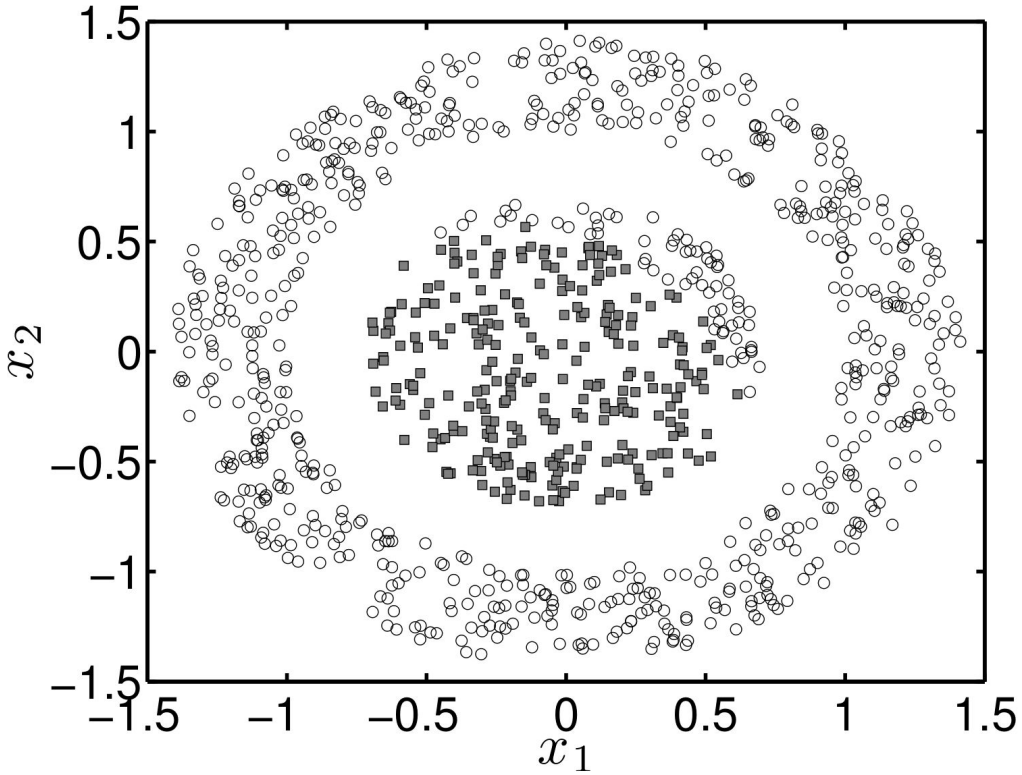
Kernel K-means - example



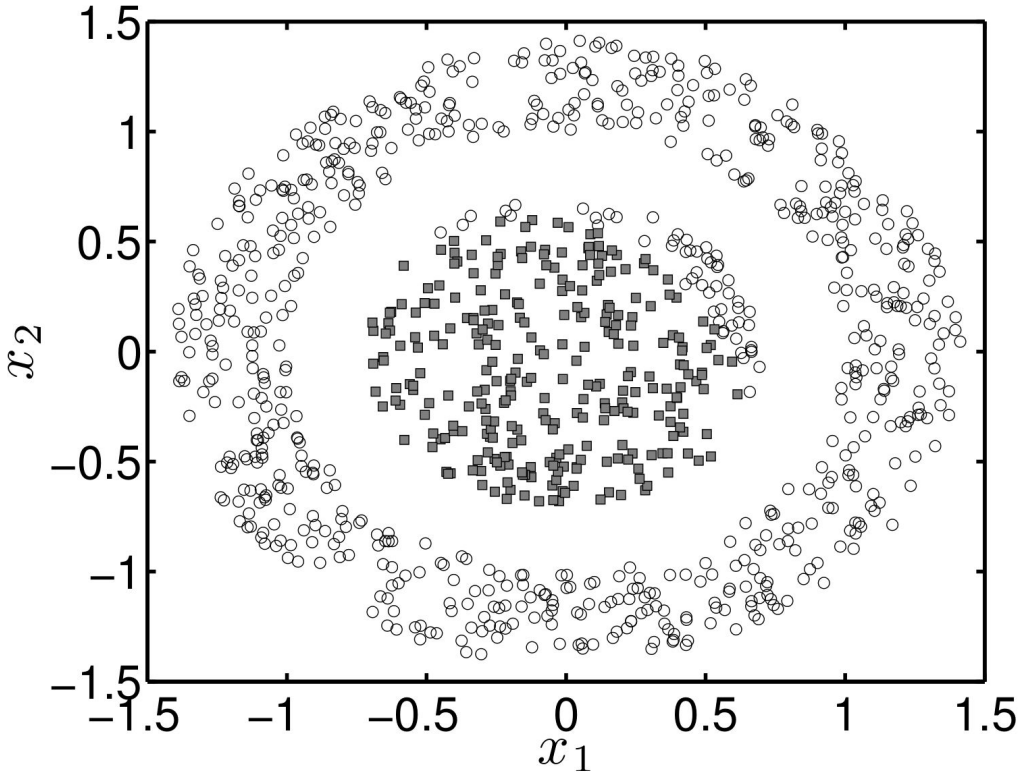
Kernel K-means - example



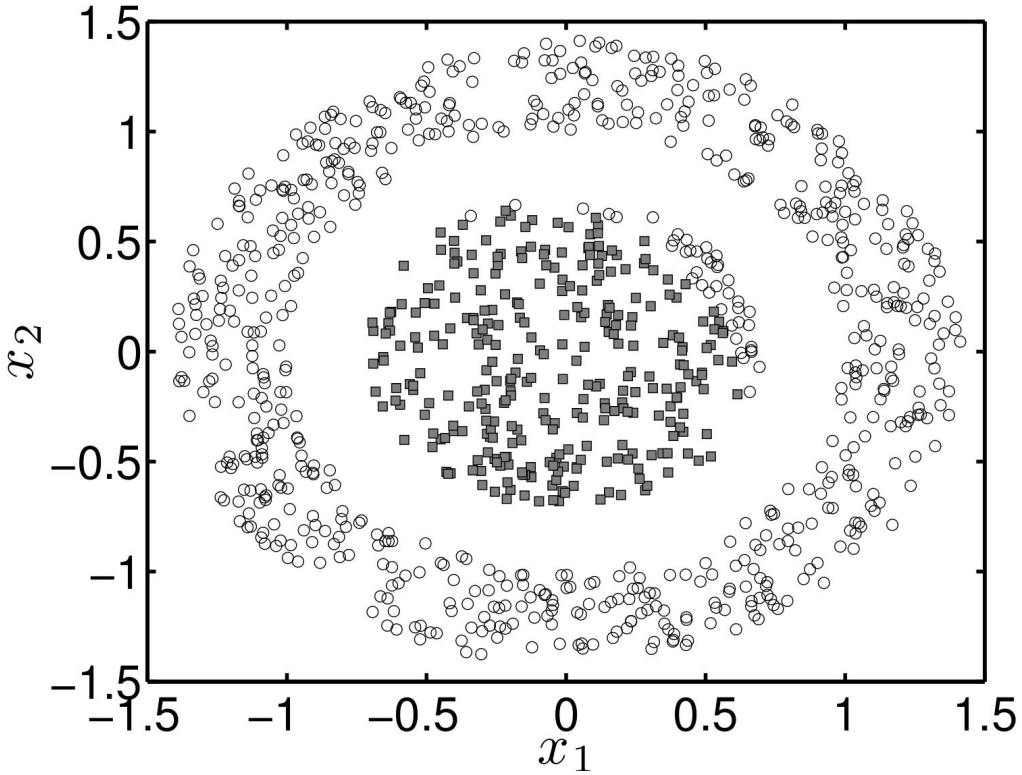
Kernel K-means - example



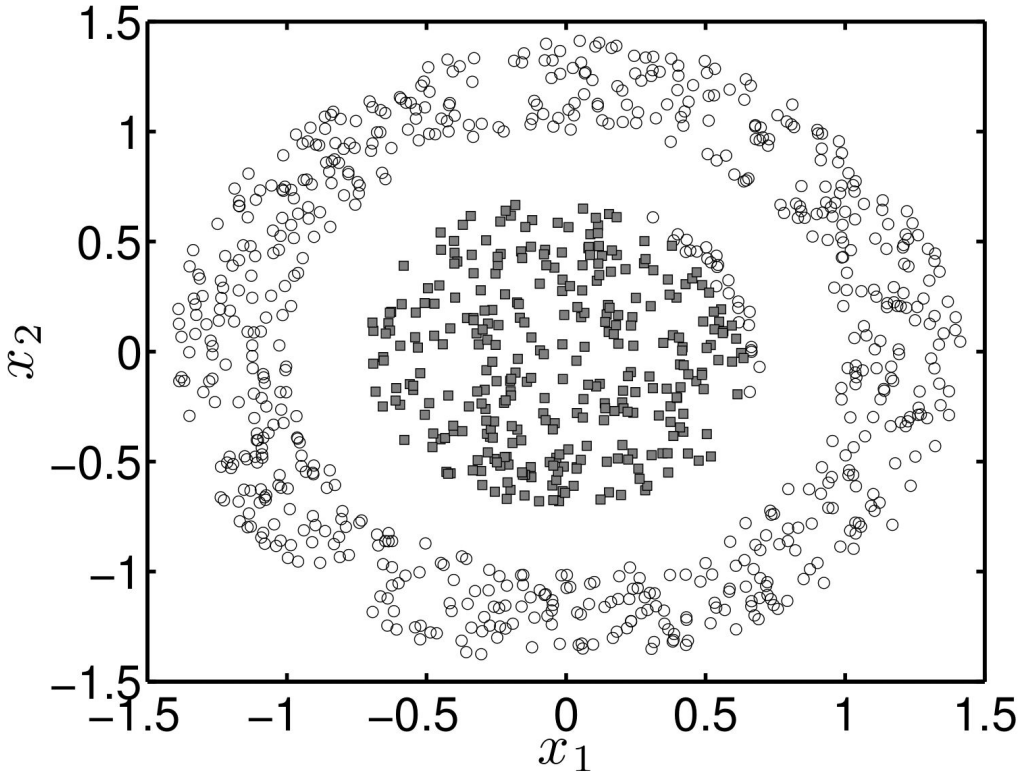
Kernel K-means - example



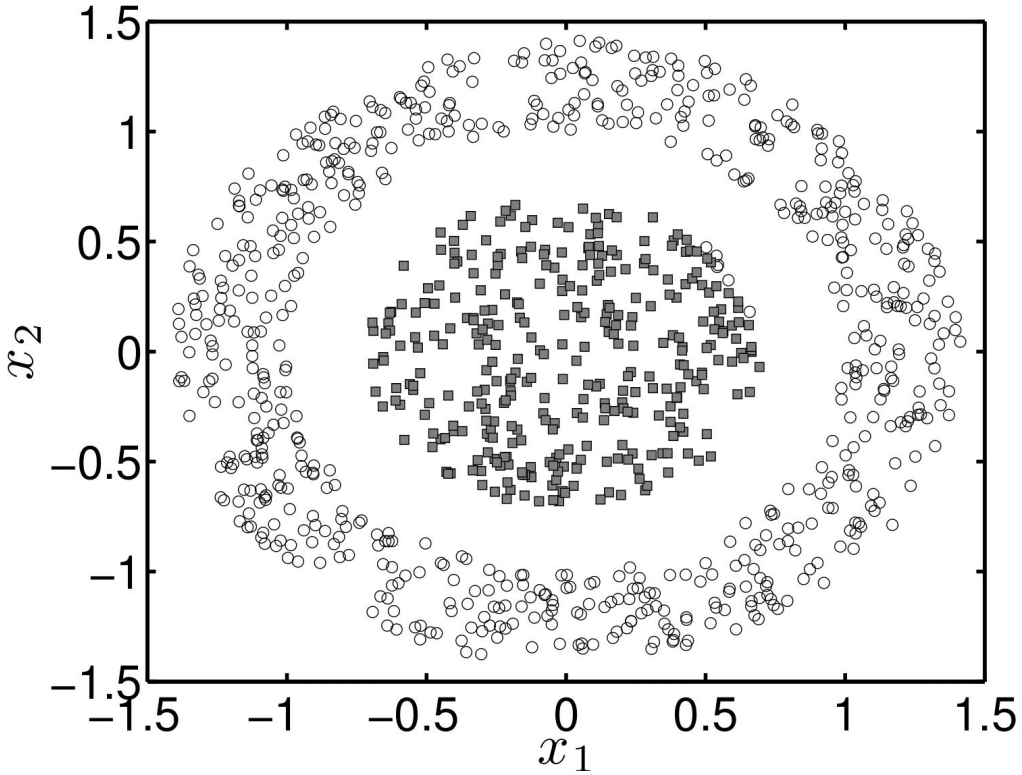
Kernel K-means - example



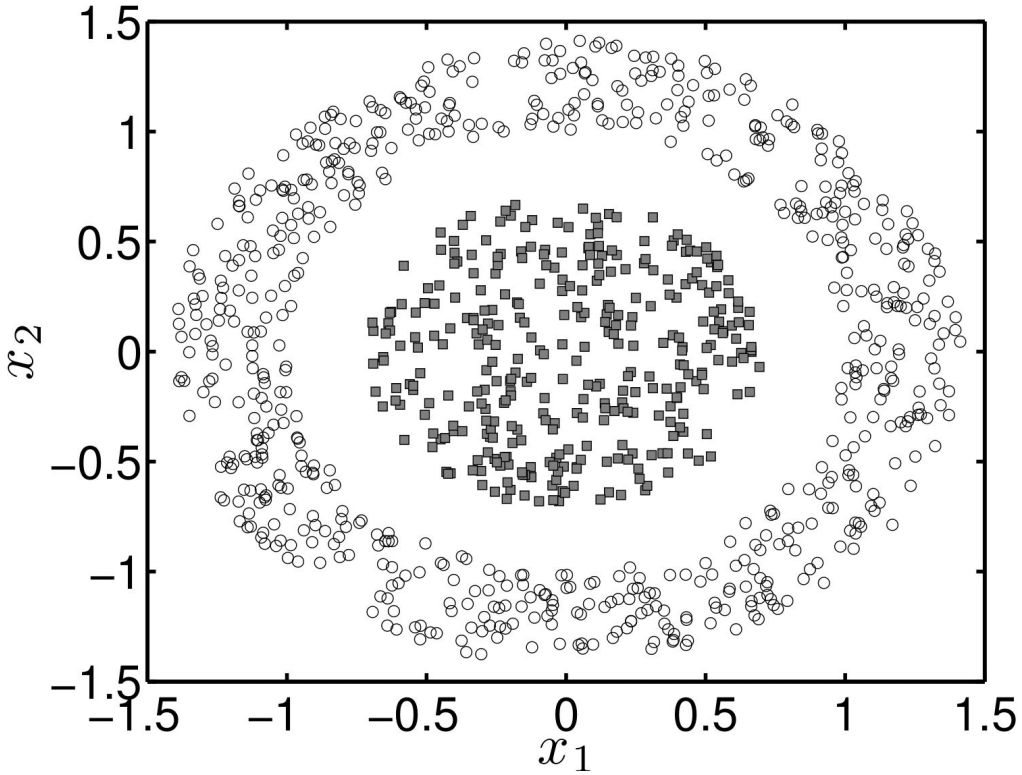
Kernel K-means - example



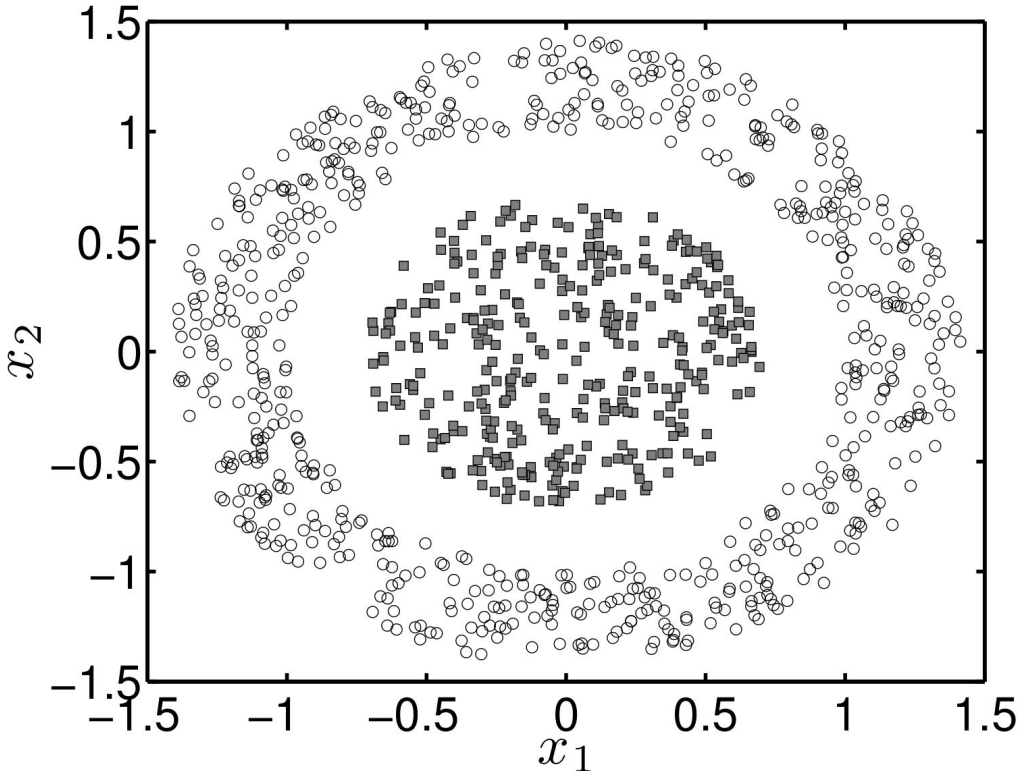
Kernel K-means - example



Kernel K-means - example



Kernel K-means - example



Kernel K-means

- Makes simple K-means algorithm more flexible.
- But, have to now set additional parameters.
- Very sensitive to initial conditions – lots of local optima.

K-means - summary

- Simple (and effective) clustering strategy.
- Converges to (local) minima of:

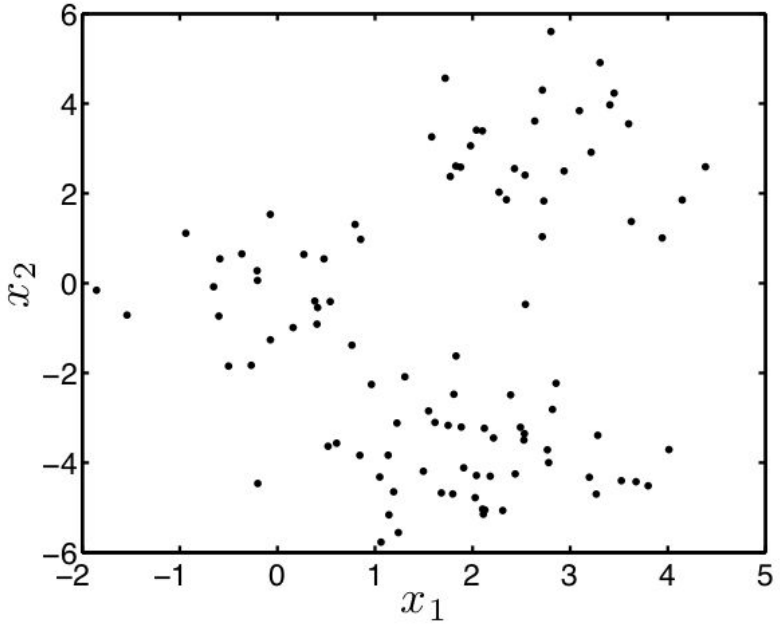
$$\sum_n \sum_k z_{nk} (\mathbf{x}_n - \boldsymbol{\mu}_k)^\mathsf{T} (\mathbf{x}_n - \boldsymbol{\mu}_k)$$

- Sensitive to initialisation.
- How do we choose K?
  - Tricky: Quantity above always decreases as K increases.
  - Can use CV if we have a measure of 'goodness'.
  - For clustering these will be application specific.

Machine Learning & Artificial Intelligence for Data Scientists: Clustering (Part 2)

Ke Yuan
School of Computing Science

Mixture models – thinking generatively



- Could we hypothesis a model that could have created this data?
- Each $\mathbf{x}_n$ seems to have come from one of three distributions.

A generative model

- Assumption:Each $\mathbf{x}_n$ comes from one of different K distributions.
- To generate $\mathbf{X}$:
- For each n:
  1. Pick one of the K components.
  2. Sample $\mathbf{x}_n$ from this distribution.

- We already have $\mathbf{X}$
- Define parameters of all these distributions as Δ.
- We'd like to reverse-engineer this process learn Δ which we can then use to find which component each point came from.
- Maximise the likelihood!

Mixture model likelihood

- Let the kth distribution have pdf:

$$p(\mathbf{x}_n|z_{nk}=1,\Delta_k)$$

- We want the likelihood:

$$p(\mathbf{X}|\Delta)$$

- First, factorise:

$$p(\mathbf{X}|\Delta)=\prod_{i=1}^{N}p(\mathbf{x}_n|\Delta)$$

- Then, un-marginalise k:

$$\begin{aligned}p(\mathbf{X}|\Delta) &= \prod_{i=1}^{N}\sum_{k=1}^{K}p(\mathbf{x}_n,z_{nk}=1|\Delta)\\ &= \prod_{i=1}^{N}\sum_{k=1}^{K}p(\mathbf{x}_n|z_{nk}=1,\Delta_k)p(z_{nk}=1|\Delta)\end{aligned}$$

Mixture model likelihood

- So, we have a likelihood:

$$p(\mathbf{X}|\Delta) = \prod_{i=1}^{N} \sum_{k=1}^{K} p(\mathbf{x}_n|z_{nk} = 1, \Delta_k) p(z_{nk} = 1|\Delta)$$

- And we want to find Δ.
- So:

$$\underset{\Delta}{\text{argmax}} \prod_{i=1}^{N} \sum_{k=1}^{K} p(\mathbf{x}_n|z_{nk} = 1, \Delta_k) p(z_{nk} = 1|\Delta)$$

- Logging made this easier before, so let's try it:

$$\underset{\Delta}{\text{argmax}} \sum_{n=1}^{N} \log \sum_{k=1}^{K} p(\mathbf{x}_n|z_{nk} = 1, \Delta_k) p(z_{nk} = 1|\Delta)$$

Gaussian mixtu model

- Assume component distributions are Gaussians with diagonal covariance:

$$p(\mathbf{x}_n|z_{nk}=1,\boldsymbol{\mu}_k,\sigma_k^2)=\mathcal{N}(\boldsymbol{\mu}_k,\sigma_k^2\mathbf{I})$$

- We need to be able to estimate the prior of assignment. Let

$$\pi_k = p(z_{nk}=1|\Delta)$$

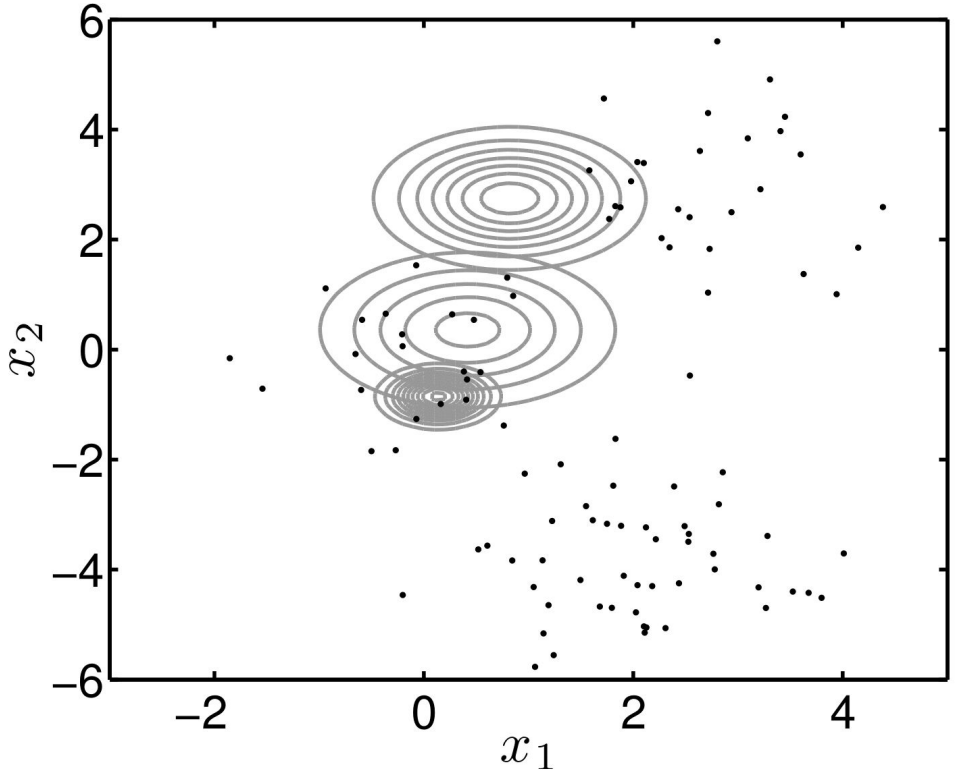
- We also want to estimate the probability to assign data to each component

$$q_{nk}=\frac{\pi_k p(\mathbf{x}_n|z_{nk}=1,\boldsymbol{\mu}_k,\sigma_k^2)}{\sum_{j=1}^{K}\pi_j p(\mathbf{x}_n|z_{nj}=1,\boldsymbol{\mu}_j,\sigma_j^2)}$$

Mixture model optimisation – the Expectation-Maximization (EM) algorithm

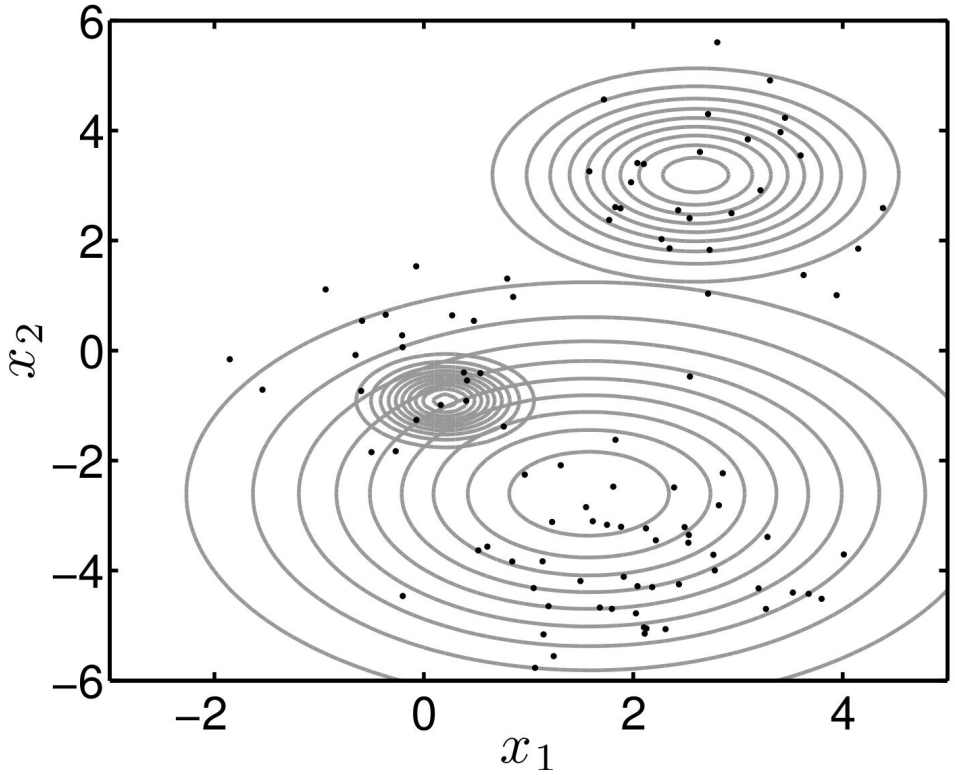
- Following optimisation algorithm:
  1. Guess $\boldsymbol{\mu}_k, \sigma_k^2, \pi_k$
  2. **(E)**xpectation-step: Compute q_{nk}
  3. **(M)**aximization-step: Update $\boldsymbol{\mu}_k, \sigma_k^2, \pi_k$
  4. Return to 2 unless parameters are unchanged.
- Guaranteed to converge to a local maximum of the lower bound.
- Note the similarity with kmeans.

Algorithm in operation



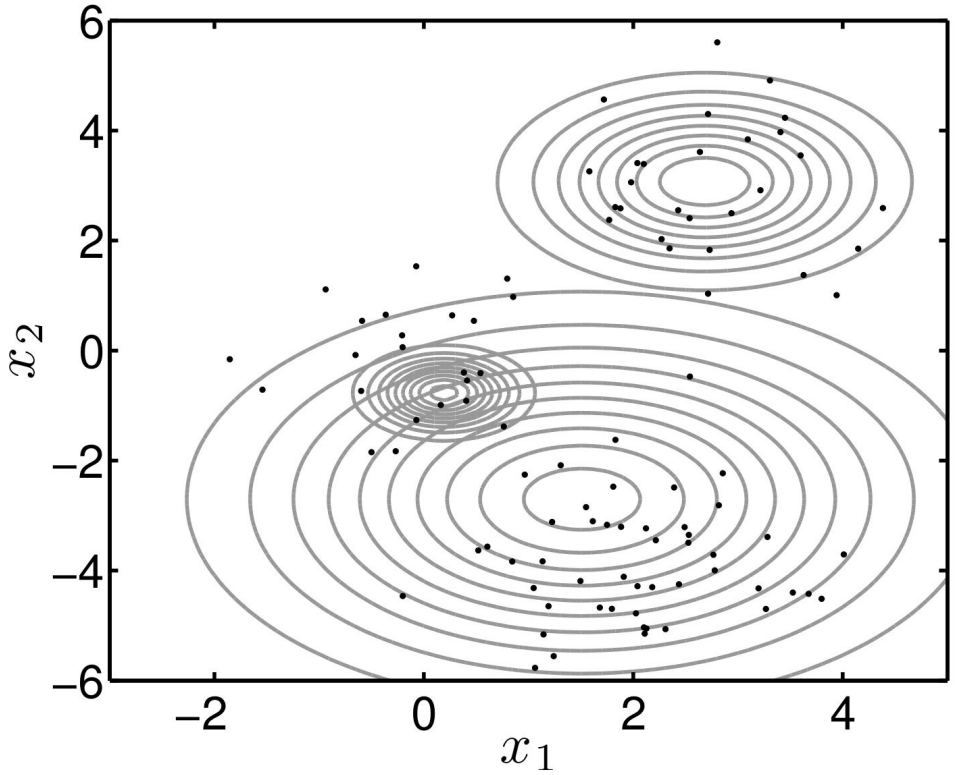
Initial guess

Algorithm in operation



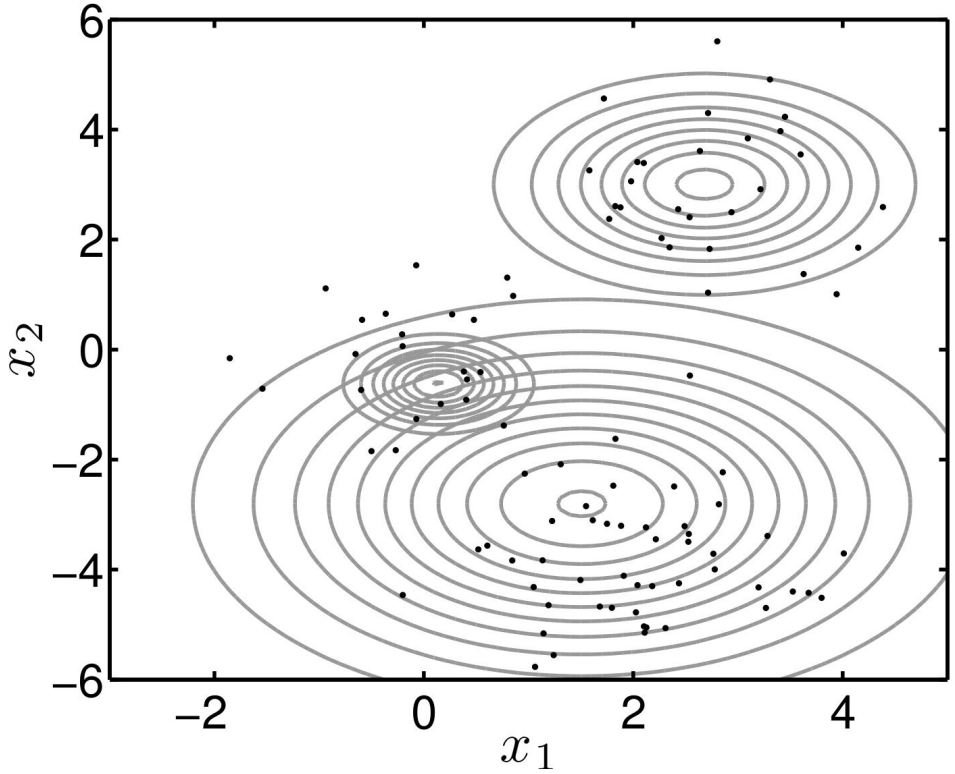
Update q_nk and then other parameters.

Algorithm in operation



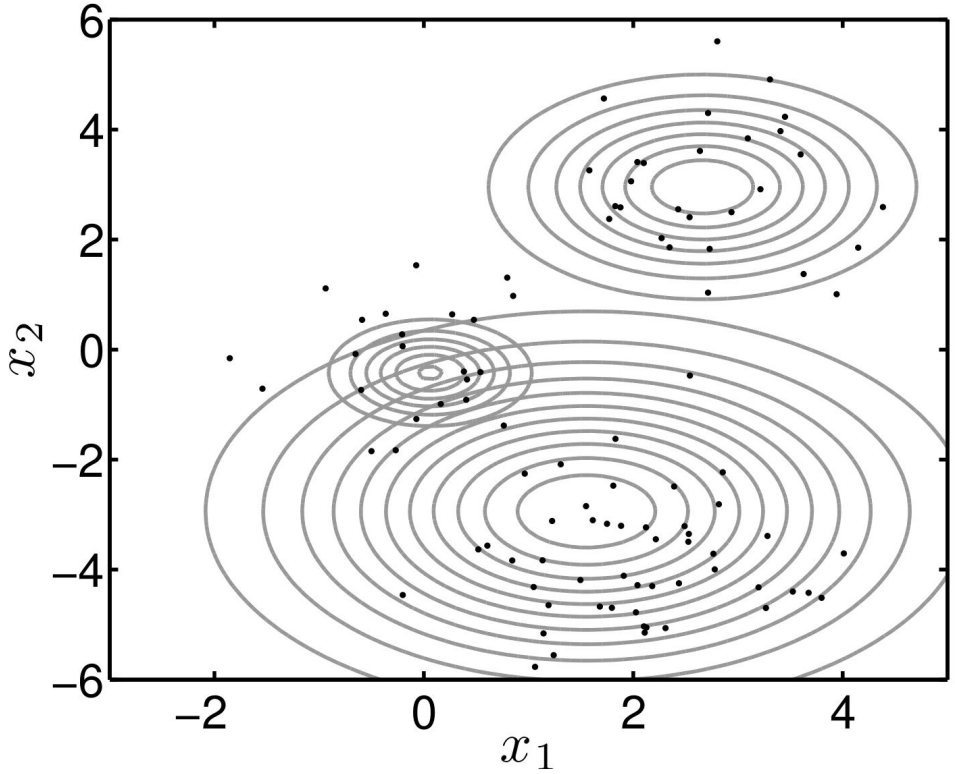
Update q_nk and then other parameters.

Algorithm in operation



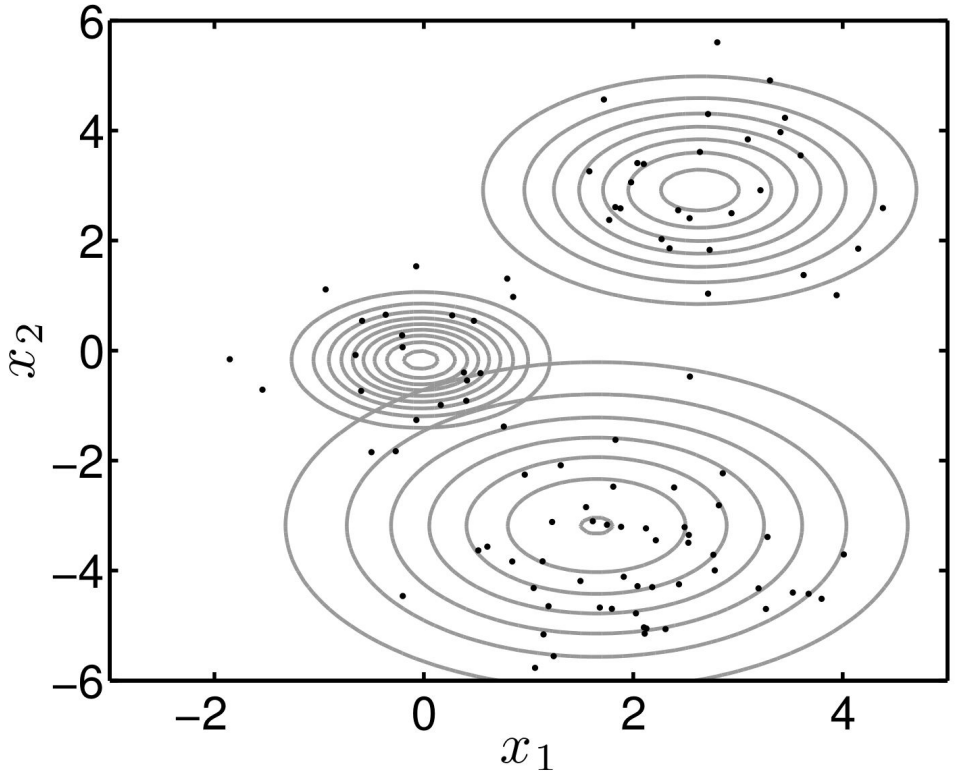
Update q_nk and then other parameters.

Algorithm in operation



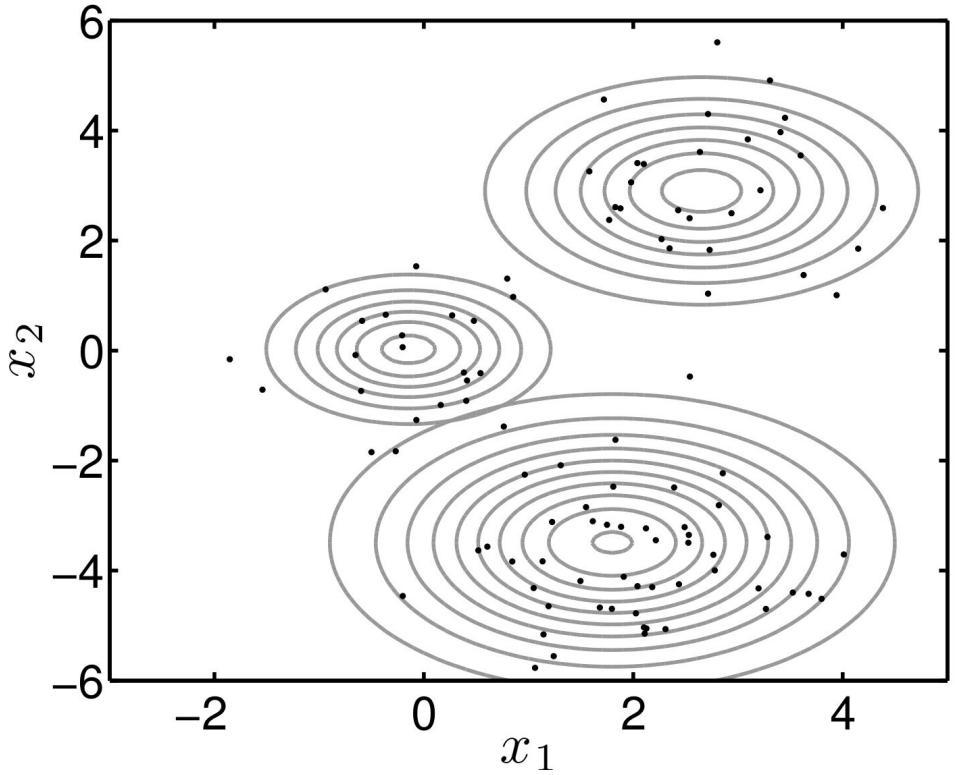
Update q_nk and then other parameters.

Algorithm in operation



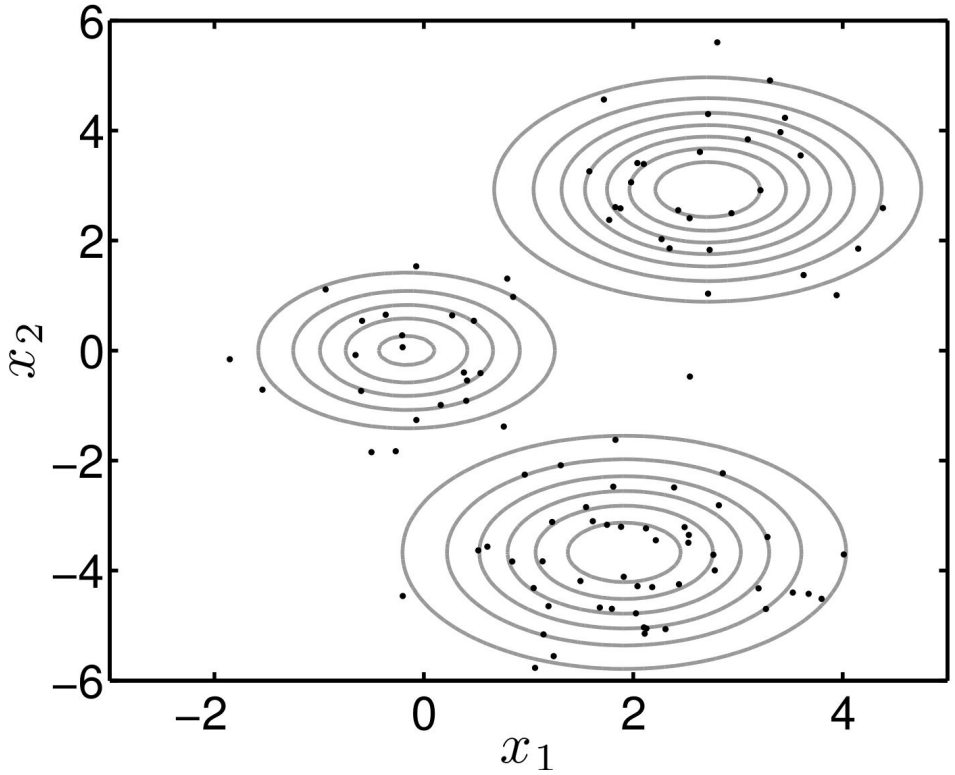
Update q_nk and then other parameters.

Algorithm in operation



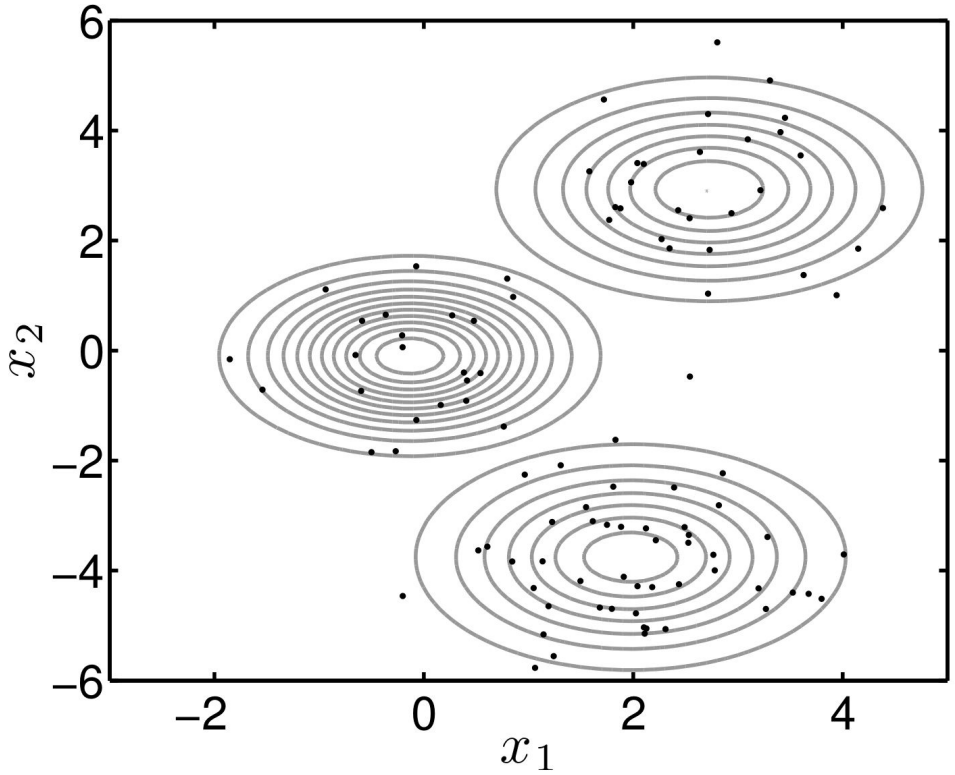
Update q_nk and then other parameters.

Algorithm in operation



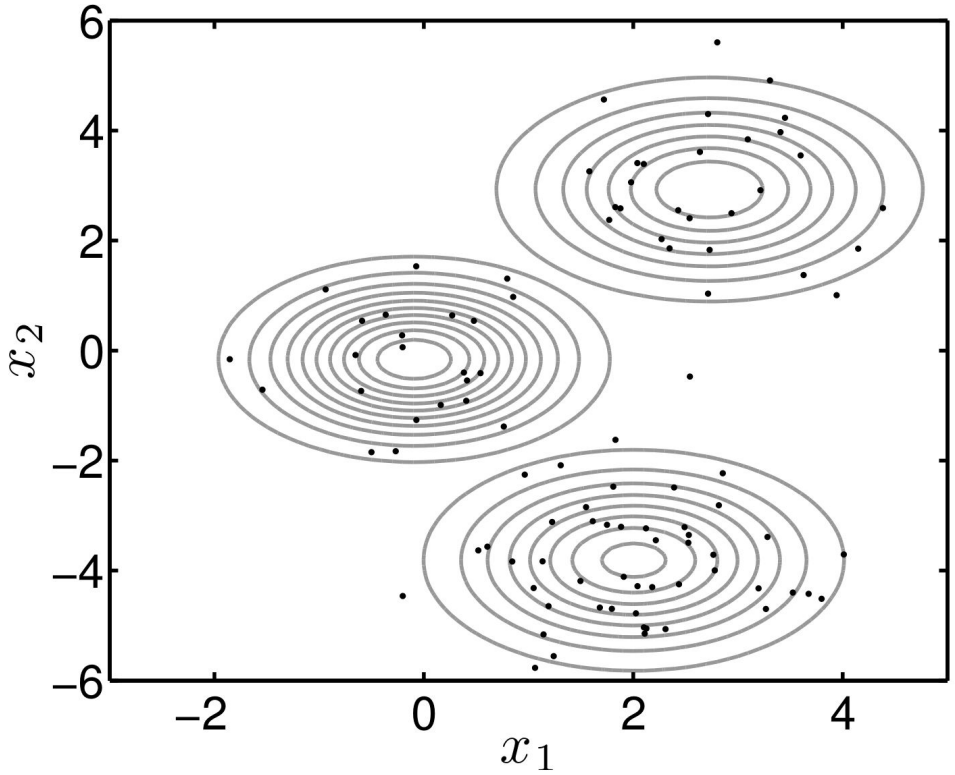
Update q_nk and then other parameters.

Algorithm in operation



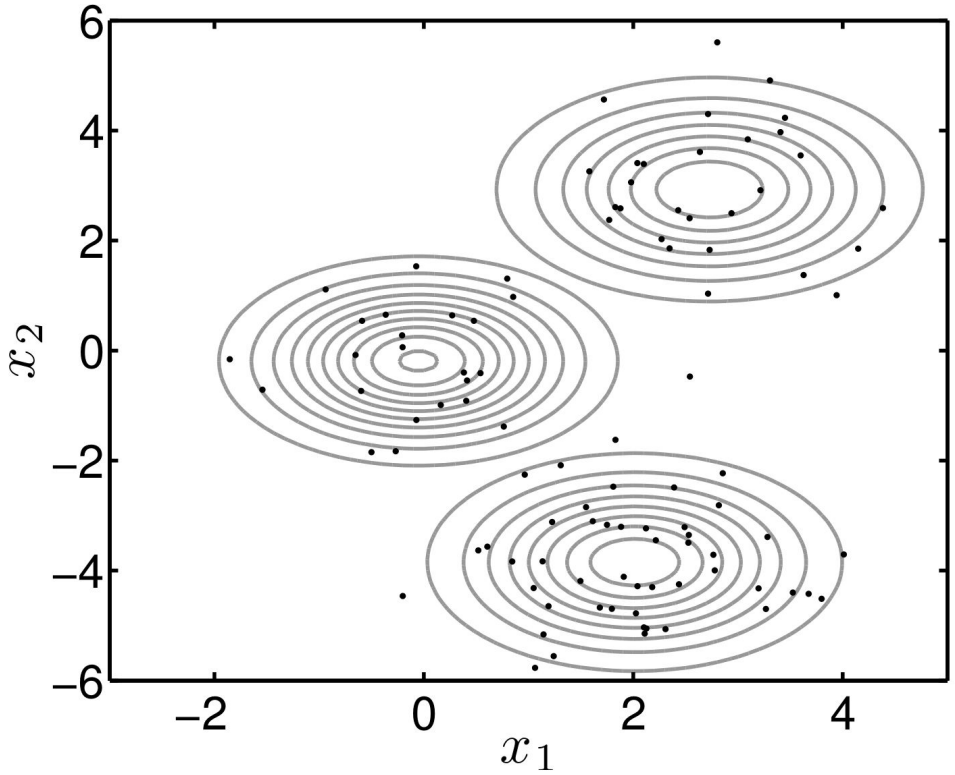
Update q_nk and then other parameters.

Algorithm in operation



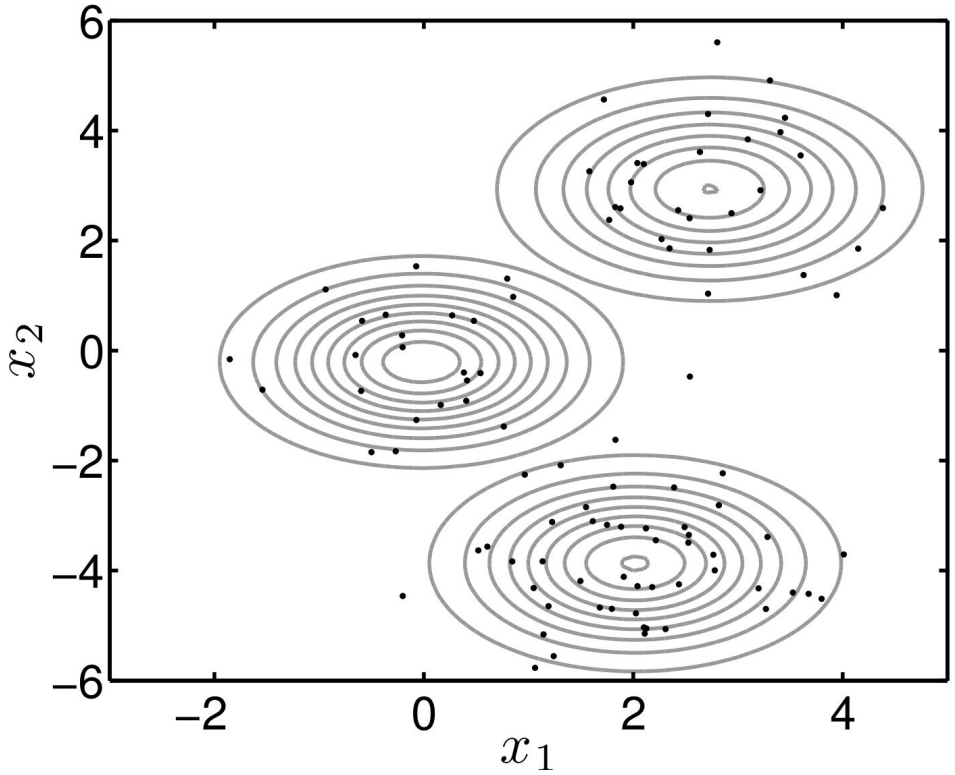
Update q_nk and then other parameters.

Algorithm in operation



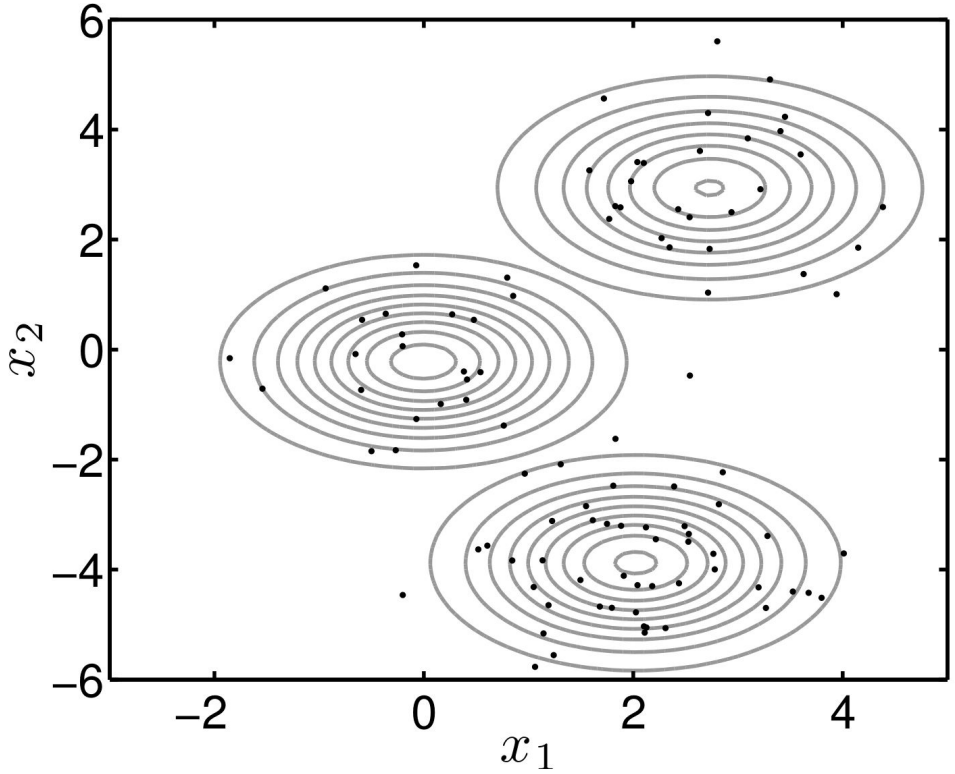
Update q_nk and then other parameters.

Algorithm in operation



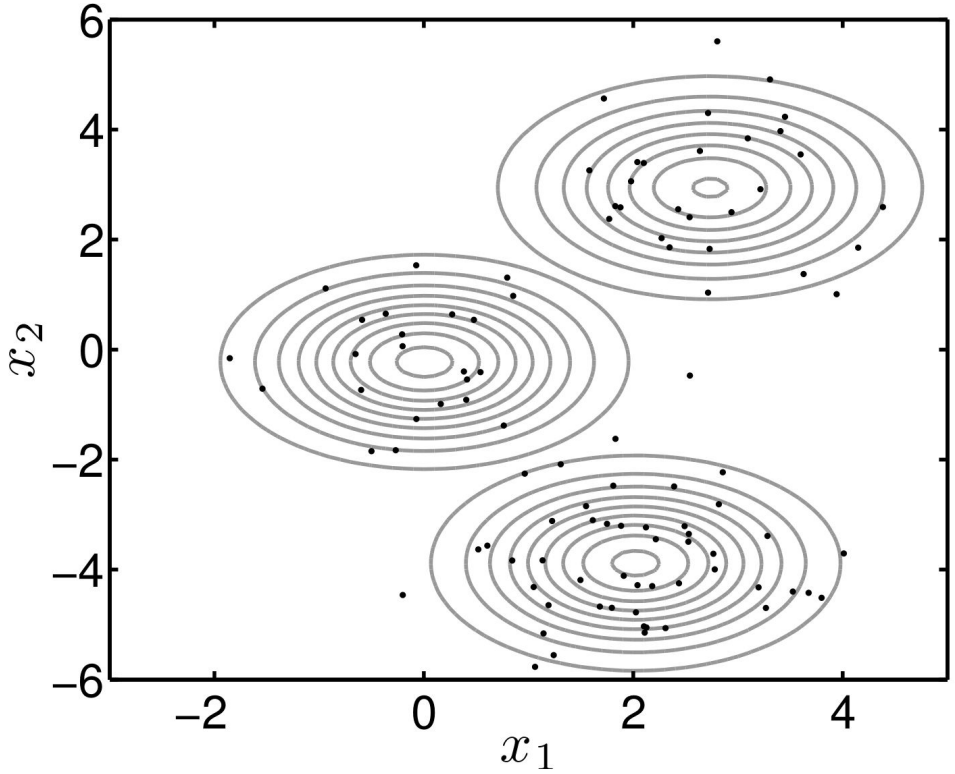
Update q_nk and then other parameters.

Algorithm in operation



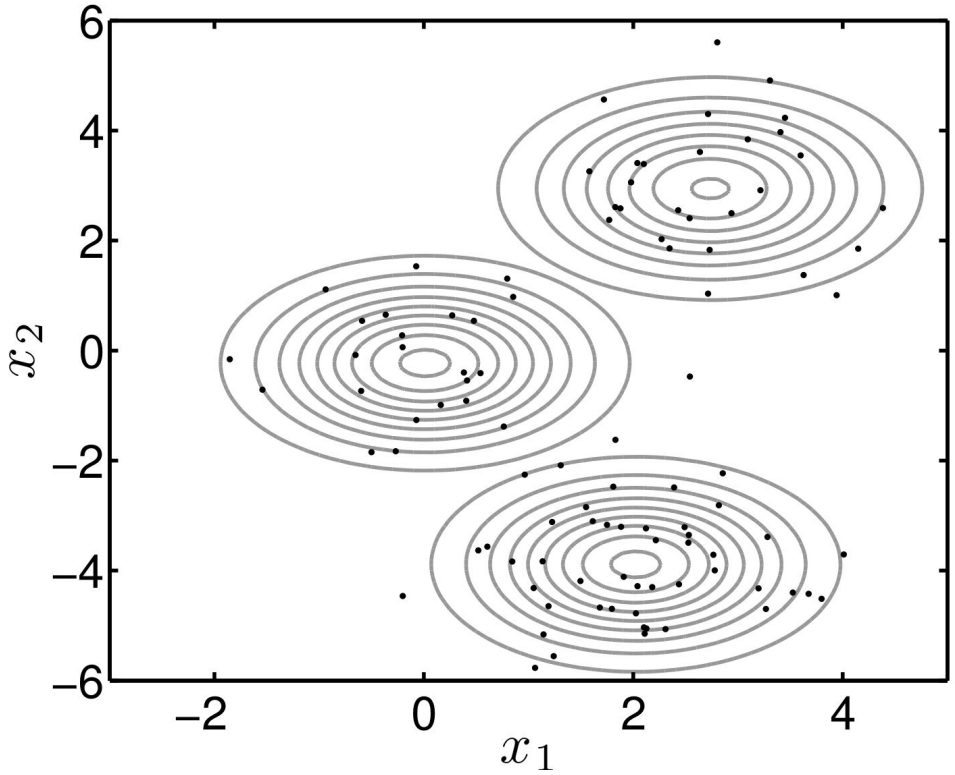
Update q_nk and then other parameters.

Algorithm in operation



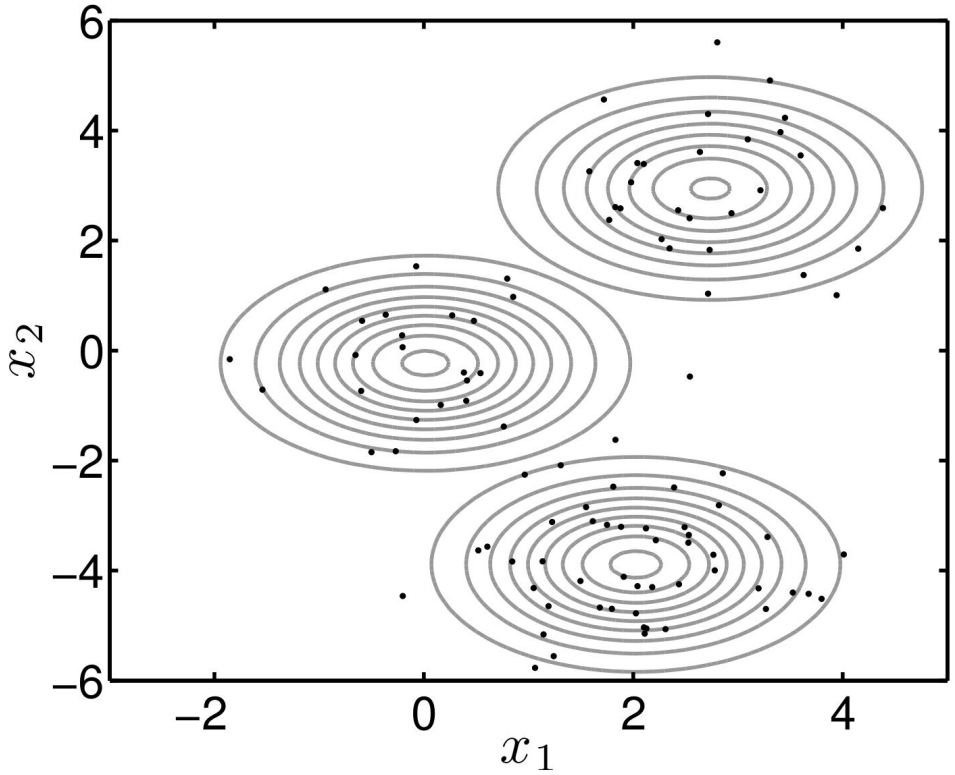
Update q_nk and then other parameters.

Algorithm in operation



Update q_nk and then other parameters.

Algorithm in operation



Solution at convergence

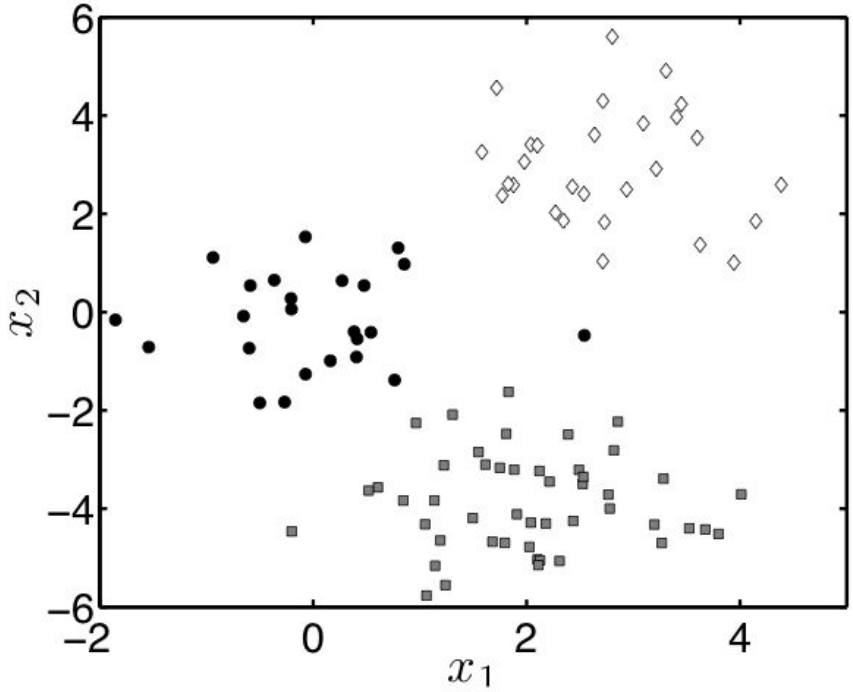
Mixture model clustering

- So, we've got the parameters, but what about the assignments?
- Which points came from which distributions?
- q_{nk} is the probability that $\mathbf{x}_n$ came from distribution k.

$$q_{nk} = P(z_{nk} = 1 | \mathbf{x}_n, \mathbf{X}, \mathbf{t})$$

- Can stick with probabilities or assign each $\mathbf{x}_n$ to it's most likely component.

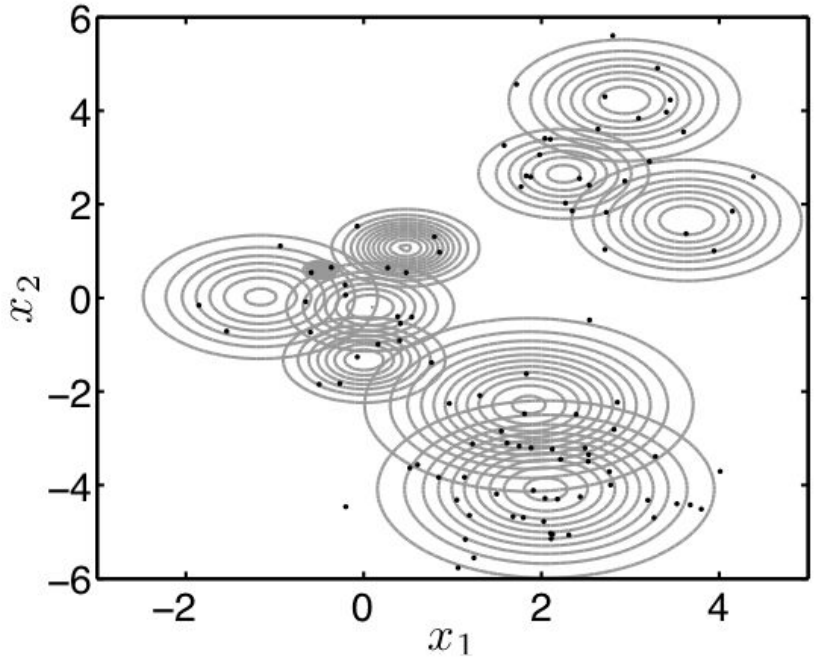
Mixture model clustering



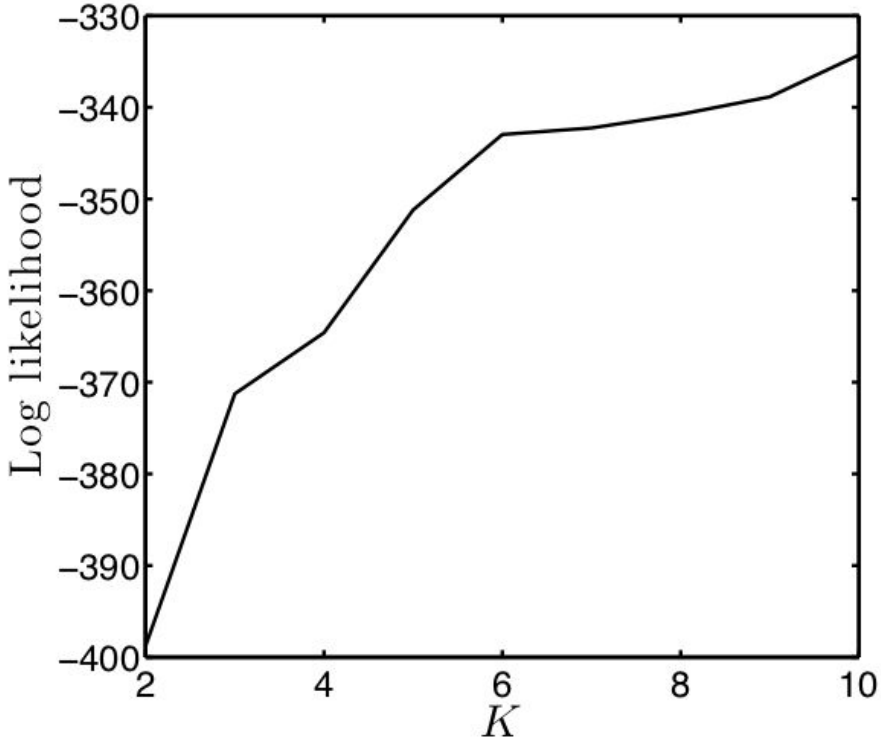
- Points assigned to the cluster with the highest q_{nk} value.

Mixture model – issues

- How do we choose K?
- What happens when we increase it?
- $K = 10$



Likelihood increase

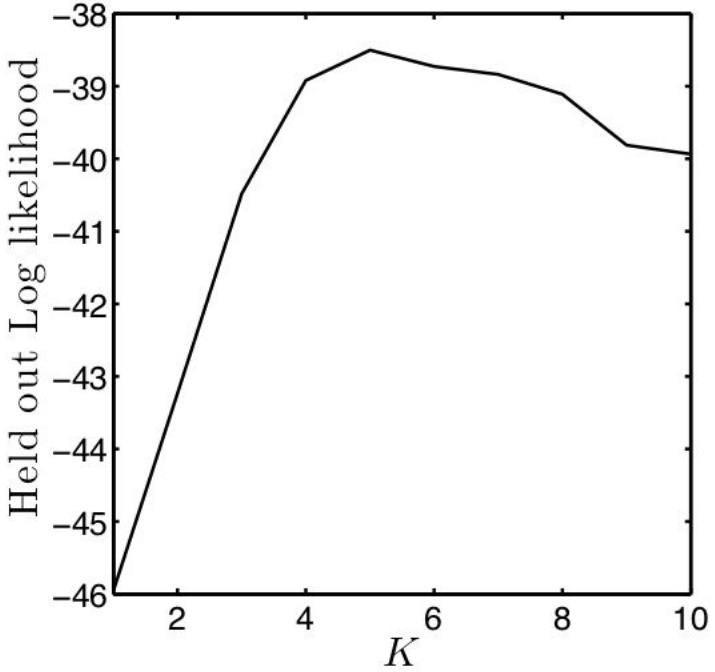


- Likelihood always increases as σ_k^2 decreases.

What can we do?

– – –

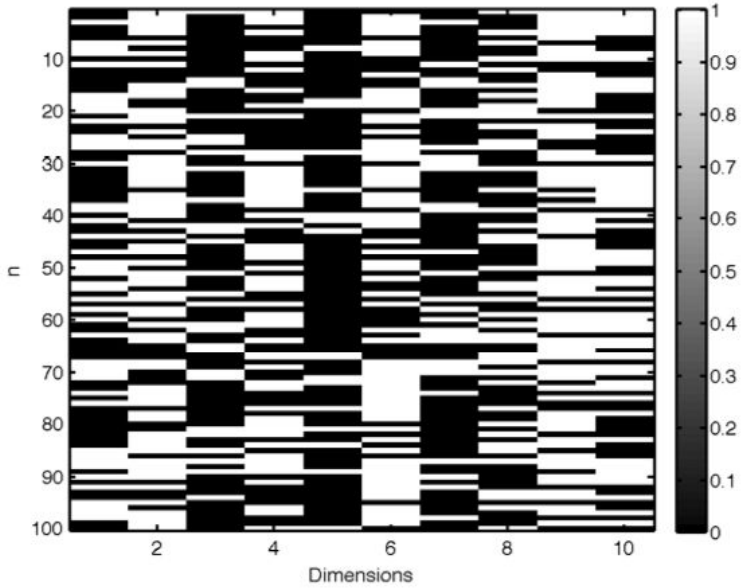
- What can we do?
- Cross-validation...



- 10-fold CV. Maximum is close to true value (3)
- 5 might be better for this data....

Mixture models – other distributions

- We've seen Gaussian distributions.
- Can actually use anything....
- As long as we can define $p(\mathbf{x}_n|z_{nk} = 1, \Delta_k)$
- e.g. Binary data:



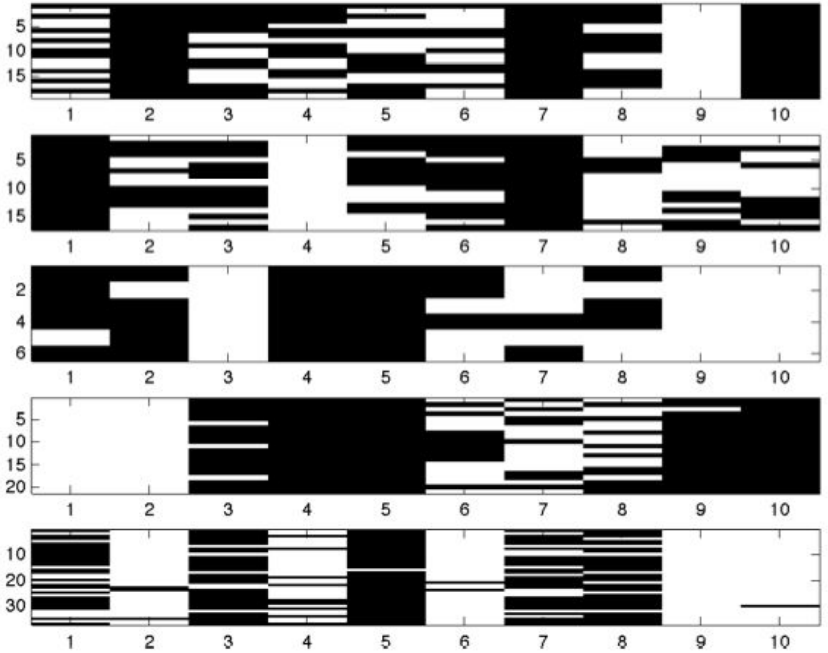
Binary example

- $\mathbf{x}_n = [0, 1, 0, 1, 1, \ldots, 0, 1]^{\mathsf{T}}$ (D dimensions)
- $p(\mathbf{x}_n | z_{nk} = 1, \Delta_k) = \prod_{d=1}^{D} p_{kd}^{x_{nd}} (1 - p_{kd})^{1 - x_{nd}}$
- Updates for p_{kd} are:

$$p_{kd} = \frac{\sum_n q_{nk} x_{nd}}{\sum_n q_{nk}}$$

- q_{nk} and π_k are the same as before...
- Initialise with random p_{kd} ($0 \leq p_{kd} \leq 1$)

Binary example



- $K = 5$ clusters.
- Clear structure present.

Summary

- Introduced two clustering methods.
- K-means
  - Very simple.
  - Iterative scheme.
  - Can be kernelised.
  - Need to choose K.
- Mixture models
  - Create a model of each class (similar to Bayes classifier)
  - Iterative sceme (EM)
  - Can use any distribution for the components.
  - Can set K by cross-validation (held-out likelihood)
  - State-of-the-art: Don't need to set K – treat as a variable in a Bayesian sampling scheme.