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

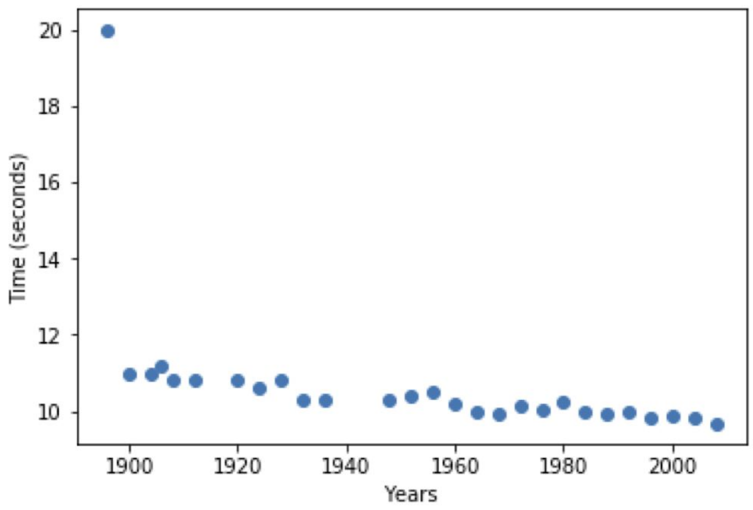
Ke Yuan
https://kyuanlab.org/
School of Computing Science

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)')
```

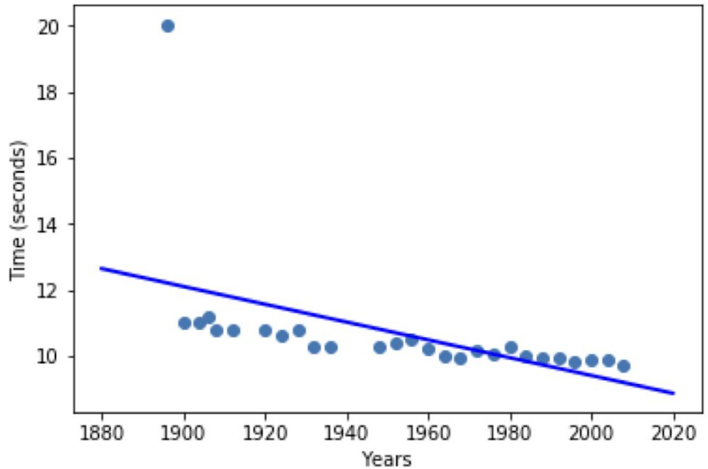


Let's add some outliers

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

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

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

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]:

```python
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]:

```python
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

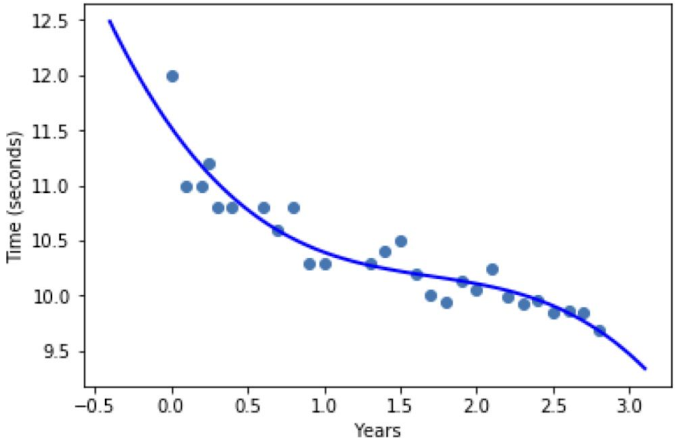
In [41]:

```
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: 0.029611321220196 76
```

Out[41]:

```
Text(0, 0.5, 'Time (seconds)')
```

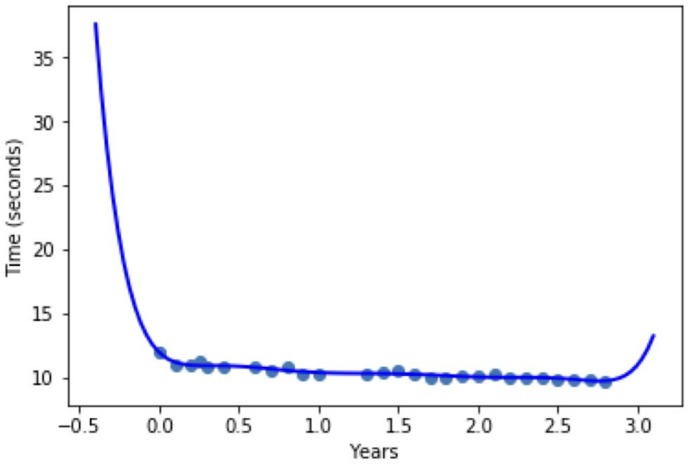


Fit the model using the same formula

In [42]:
```python
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)')
```



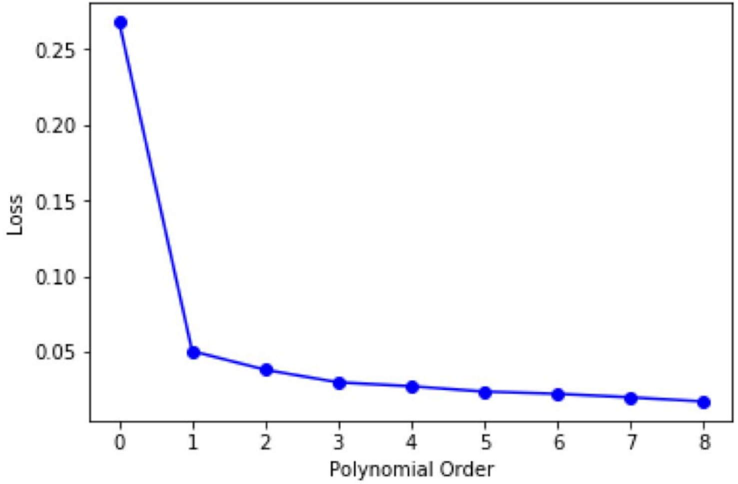
What about higher order?

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
```

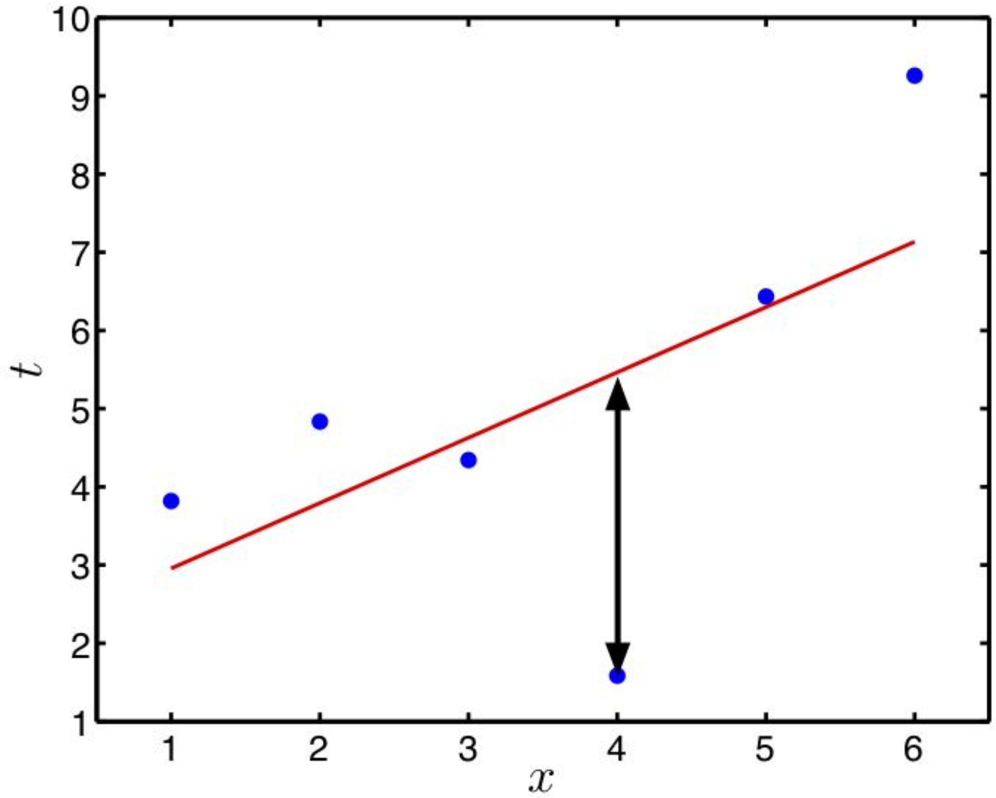
Out[84]:

```
Text(0, 0.5, 'Loss')
```



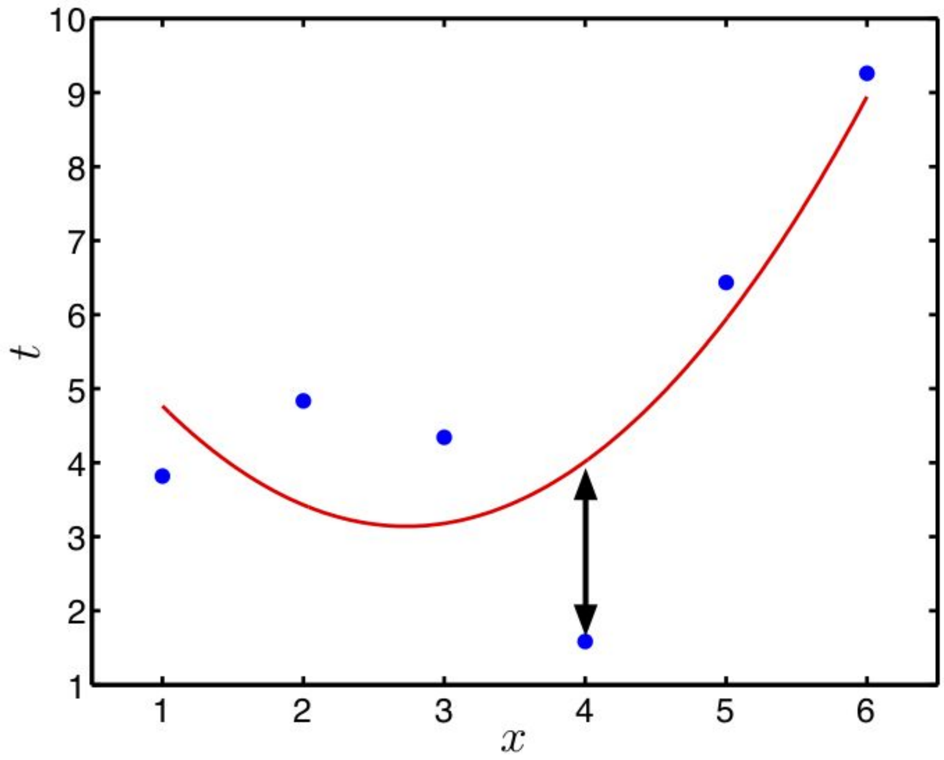
Loss always decreases as the model is made more complex

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



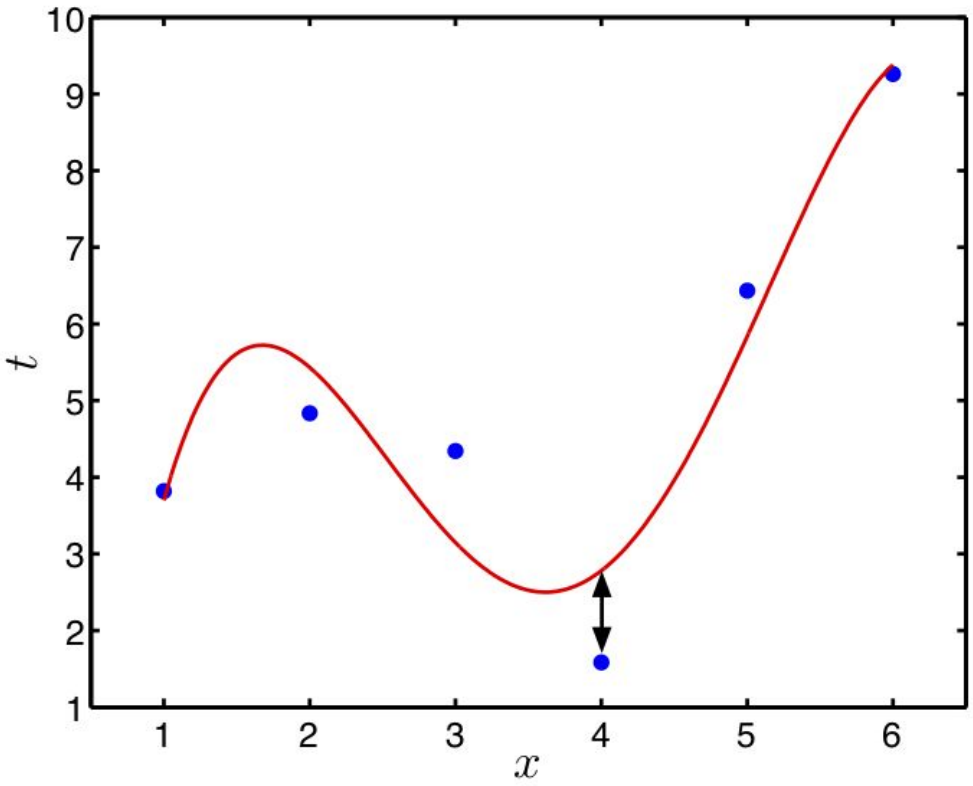
Linear model $t = w_0 + w_1 x$.

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



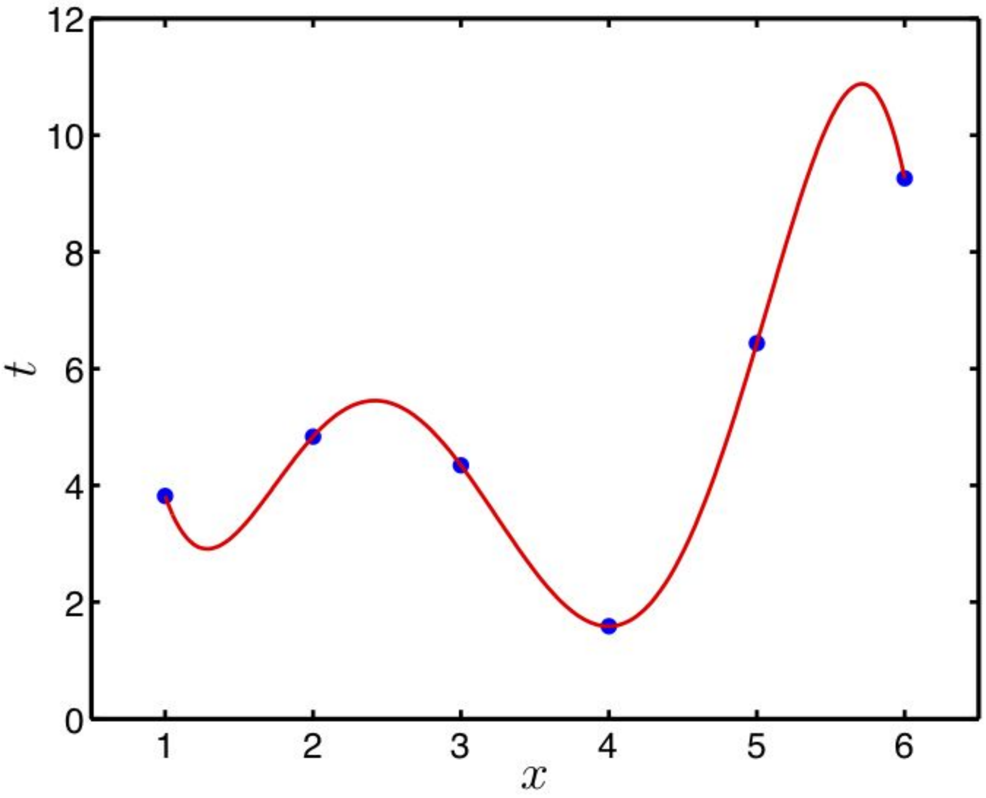
Quadratic model $t = w_0 + w_1 x + w_2 x^2$.

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:

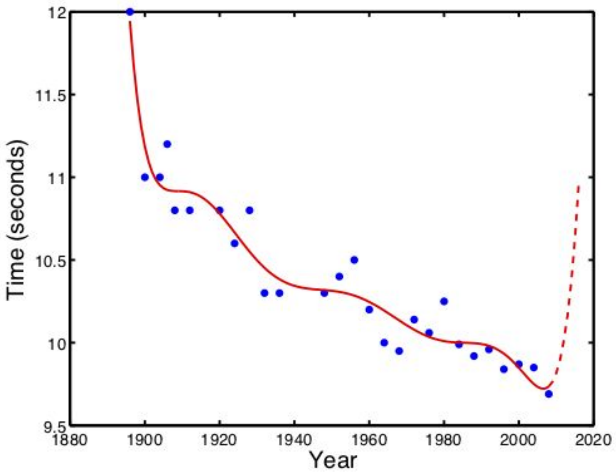


Fifth order $t = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + w_5 x^5$.

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.

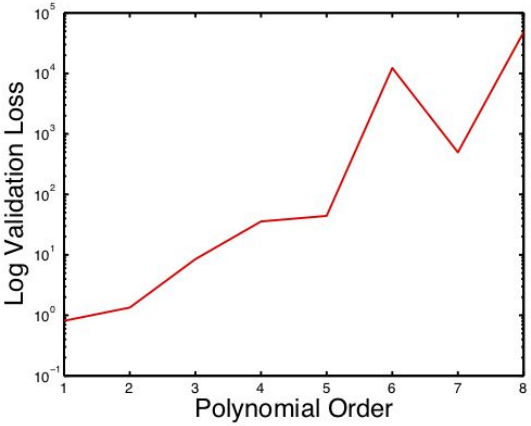
How do we choose the right model complexity? Where can we get more data?

- 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



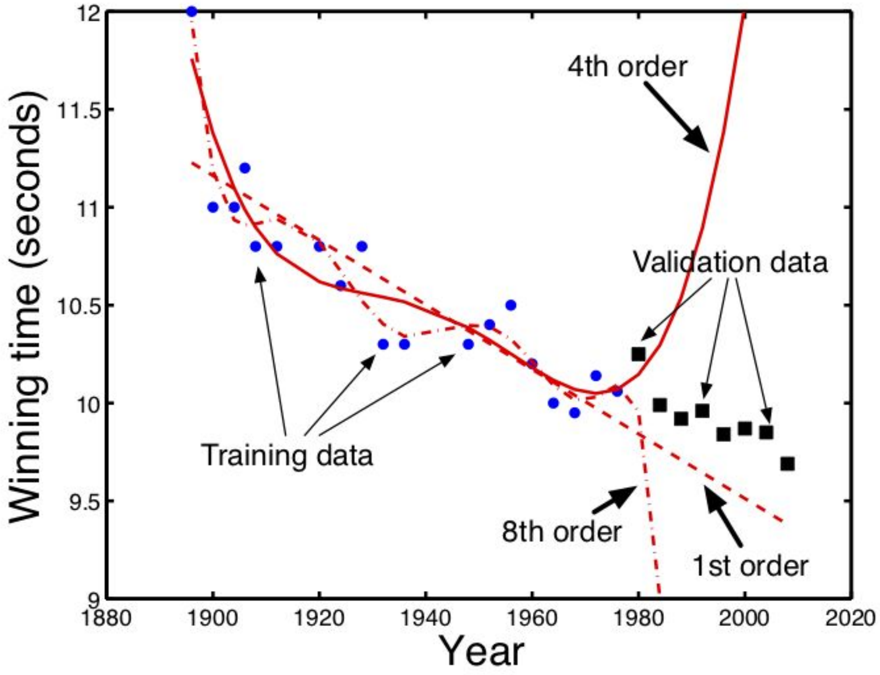
Predictions evaluated using validation loss:

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

Best model?

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

Validation example

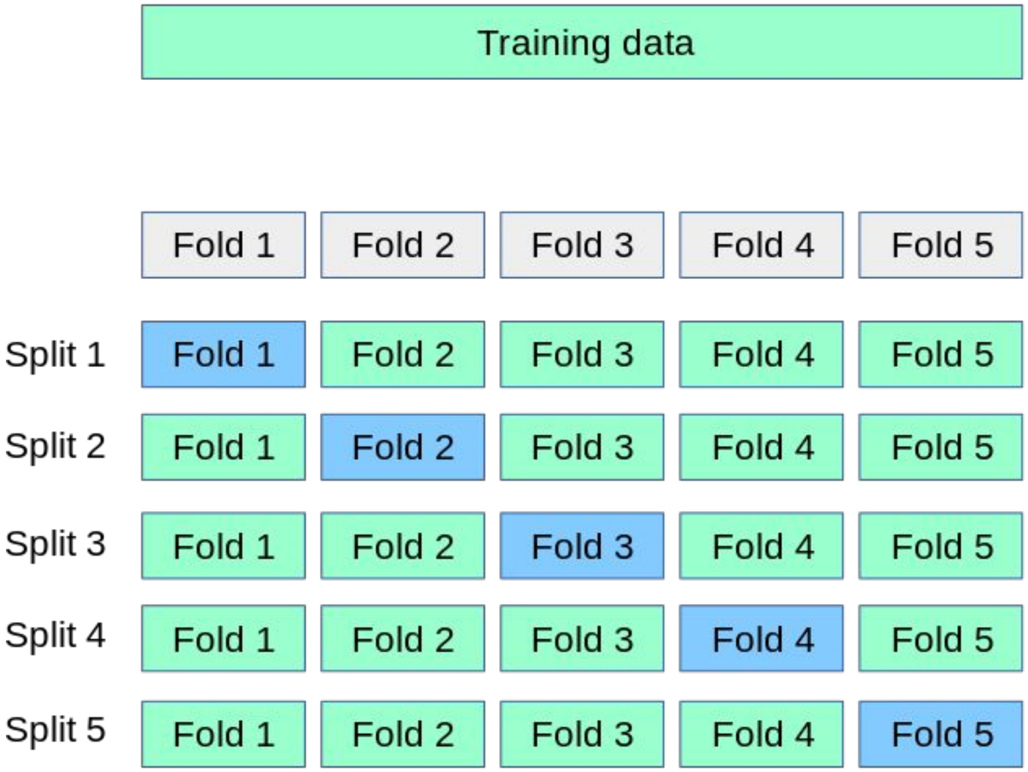


Best model

First order (linear) model generalises best.

Cross-validation (CV)

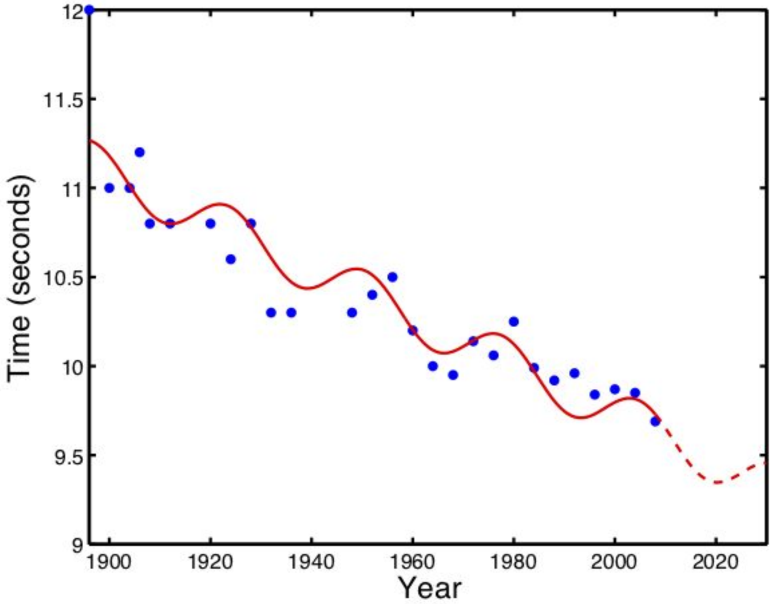
- **Cross-validation can be repeated to make results more accurate.**
- e.g. Doing 5-fold CV 10 times gives us 50 performance values to average over.
- Extreme example is when C = N so each fold includes one input-response pair: **Leave-one-out (LOO) CV**.



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

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

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, 0.036850905476505554, 0.00851823204
46148, 0.09683174323737713]
0.05246157157995471 5
```

5-fold CV loss at order 8

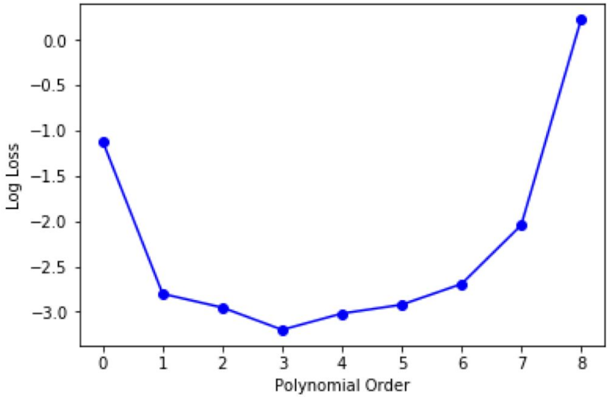
In [206]:

```
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')
```



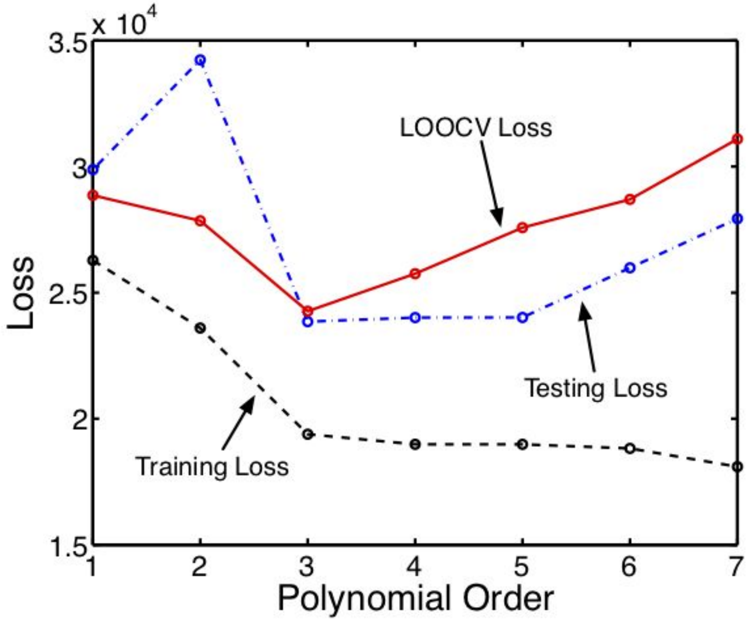
10 fold CV at polynomial order 0 to 8

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

- 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:



(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

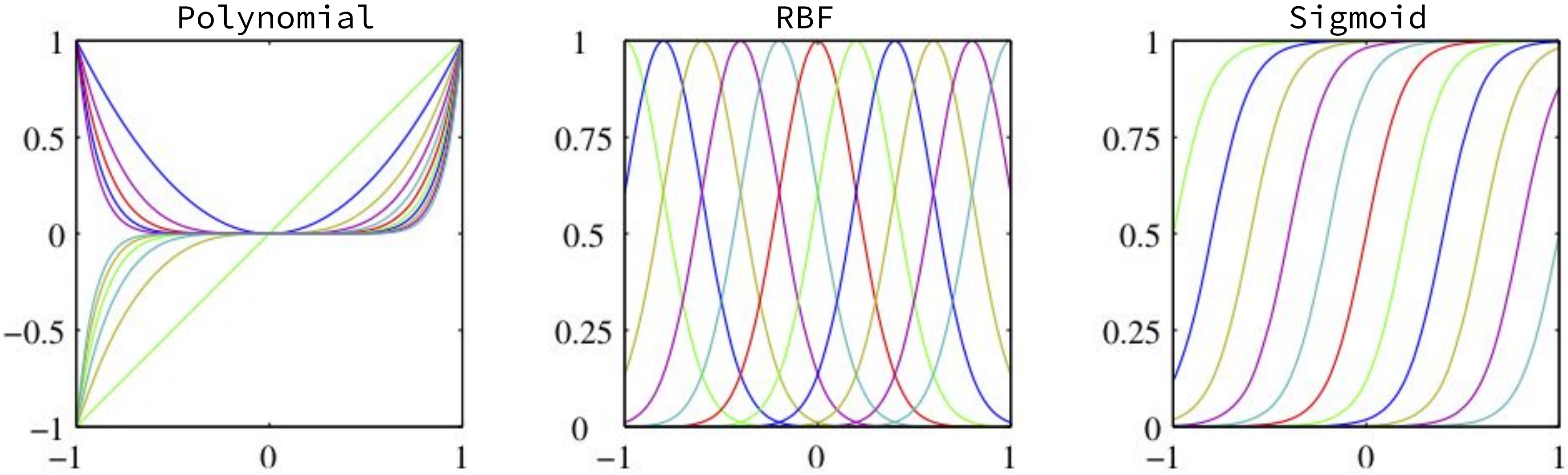
Radial basis function (RBF)

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

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.