# Machine Learning

Linear Regression

# Agenda

- Single Dimension Linear Regression
- Multi Dimension Linear Regression
- Gradient Descent
- Generalisation, Over-fitting & Regularisation
- Categorical Inputs

# What is Linear Regression?

- Learning
  - A supervised algorithm that learns from a set of training samples.
  - Each training sample has one or more input values and a single output value.
  - The algorithm learns the line, plane or hyper-plane that best fits the training samples.
- Prediction
  - Use the learned line, plane or hyper-plane to predict the output value for any input sample.

# Single Dimension Linear Regression

# Single Dimension Linear Regression

- Single dimension linear regression has pairs of *x* and *y* values as input training samples.
- It uses these training sample to derive a line that predicts values of *y*.
- The training samples are used to derive the values of *a* and *b* that minimise the error between actual and predicated values of *y*.

$$\hat{y} = ax + b$$

# Single Dimension Linear Regression

- We want a line that minimises the error between the Y values in training samples and the Y values that the line passes through.
- Or put another way, we want the line that “best fits’ the training samples.
- So we define the error function for our algorithm so we can minimise that error.

$$E = \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

# Single Dimension Linear Regression

- To determine the value of *a* that minimises the error *E*, we look for where the partial differential of *E* with respect to *a* is zero.

$$\frac{\partial E}{\partial a} = 0$$

$$\sum_{i=1}^{n} 2(y_i - \hat{y_i}) \frac{\partial \hat{y_i}}{\partial a} = 0$$

$$\sum_{i=1}^{n} 2(y_i - \hat{y_i})(-x_i) = 0$$

$$\sum_{i=1}^{n} 2(y_i - \hat{y_i})x_i = 0$$

$$\sum_{i=1}^{n} y_i x_i = \sum_{i=1}^{n} \hat{y_i} x_i$$

$$\sum_{i=1}^{n} y_i x_i = \sum_{i=1}^{n} (ax_i + b)x_i$$

$$\sum_{i=1}^{n} y_i x_i = a \sum_{i=1}^{n} ax_i^2 + b \sum_{i=1}^{n} x_i$$

# Single Dimension Linear Regression

- To determine the value of *b* that minimises the error *E*, we look for where the partial differential of *E* with respect to *b* is zero.

$$\frac{\partial E}{\partial b} = 0$$

$$\sum_{i=1}^{n} 2(y_i - \hat{y_i}) \frac{\partial \hat{y_i}}{\partial b} = 0$$

$$\sum_{i=1}^{n} (y_i - \hat{y_i}) \frac{\partial \hat{y_i}}{\partial b} = 0$$

$$\sum_{i=1}^{n} (y_i) = \sum_{i=1}^{n} (ax_i + b)$$

$$\sum_{i=1}^{n} y_i = a \sum_{i=1}^{n} x_i + bn$$

$$\frac{\sum_{i=1}^{n} y_i}{n} = \frac{a \sum_{i=1}^{n} x_i}{n} + b$$

# Single Dimension Linear Regression

- By substituting the final equations from the previous two slides we derive equations for a and b that minimise the error

$$a = \frac{\sum_i y_i x_i - \bar{y} \sum_i x_i}{\sum_i x_i^2 - \bar{x} \sum_i x_i}$$

$$b = \frac{\bar{y} \sum_i x_i^2 - \bar{x} \sum_i y_i x_i}{\sum_i x_i^2 - \bar{x} \sum_i x_i}$$

# Single Dimension Linear Regression

- We also define a function which we can use to score how well derived line fits.
- A value of 1 indicates a perfect fit.
- A value of 0 indicates a fit that is no better than simply predicting the mean of the input y values.
- A negative value indicates a fit that is even worse than just predicting the mean of the input y values.

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y})^2}{\sum_i (y_i - \bar{y})^2}$$

# Single Dimension Linear Regression

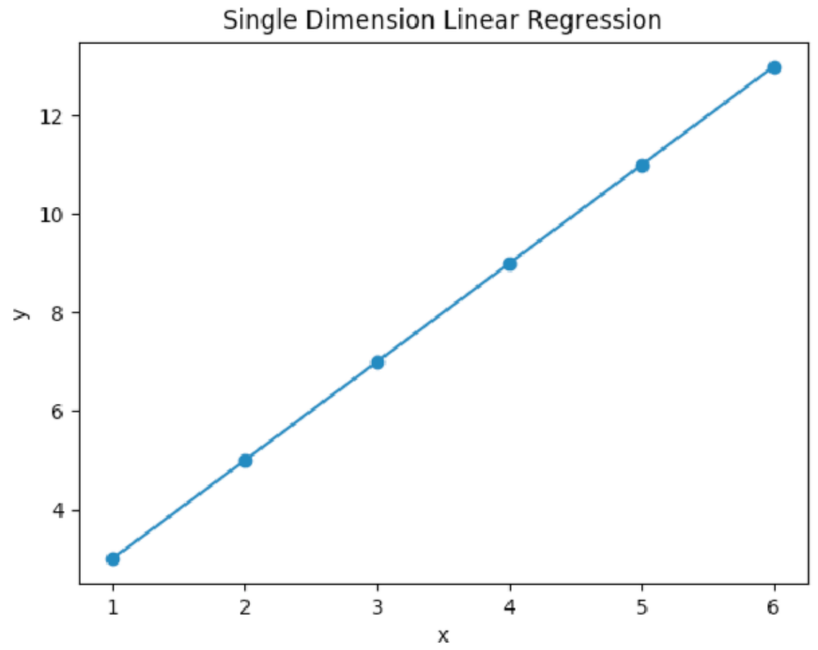
```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...:

In [2]: # Create some training samples.
   ...: X = np.array([1, 2, 3, 4, 5, 6])
   ...: Y = np.array([3, 5, 7, 9, 11, 13])
   ...:

In [3]: # Calculate values of a and b that gives least error.
   ...: denominator = X.dot(X) - X.mean() * X.sum()
   ...: a = (X.dot(Y) - Y.mean() * X.sum()) / denominator
   ...: b = (Y.mean() * X.dot(X) - X.mean() * X.dot(Y)) / denominator
   ...:

In [4]: a
Out[4]: 2.0

In [5]: b
Out[5]: 1.0
```

# Single Dimension Linear Regression

```
In [6]: # Calculate predictes values of Y from the line equation with a and b.
   ...: Yhat = a * X + b

In [7]: Yhat
Out[7]: array([  3.,   5.,   7.,   9.,  11.,  13.])

In [8]: # Calculate an score of the
   ...: d1 = Y - Yhat
   ...: d2 = Y - Y.mean()
   ...: rsquared = 1 - d1.dot(d1) / d2.dot(d2)
   ...:

In [9]: rsquared
Out[9]: 1.0
```

# Single Dimension Linear Regression

```
In [10]: plt.scatter(X, Y)
    ...: plt.plot(X, Yhat)
    ...: plt.title("Single Dimension Linear Regression")
    ...: plt.xlabel("x")
    ...: plt.ylabel("y")
    ...: plt.show()
    ...:
```



# Multi Dimension Linear Regression

# Multi Dimension Linear Regression

- Each training sample has an *x* made up of multiple input values and a corresponding *y* with a single value.
- The inputs can be represented as an *X* matrix in which each row is sample and each column is a dimension.
- The outputs can be represented as *y* matrix in which each row is a sample.

$$X = \begin{bmatrix} x_{1,1} & x_{1,\dots} & x_{1,d} \\ x_{\dots,1} & x_{\dots,\dots} & x_{\dots,d} \\ x_{n,1} & x_{n,\dots} & x_{n,d} \end{bmatrix}$$

$$X_{ij} \rightarrow sample_i, dimension_j$$

$$y = \begin{bmatrix} y_1 \\ y_{\dots} \\ y_n \end{bmatrix}$$

# Multi Dimension Linear Regression

- Our predicated y values are calculated by multiple the X matrix by a matrix of weights, w.

- If there are 2 dimension, then this equation defines plane. If there are more dimensions then it defines a hyper-plane.

$$\hat{y} = Xw$$

$$w = \begin{bmatrix} w_1 \\ w_{\dots} \\ w_d \end{bmatrix}$$

# Multi Dimension Linear Regression

- We want a plane or hyper-plane that minimises the error between the *y* values in training samples and the *y* values that the plane or hyper-plane passes through.

- Or put another way, we want the plane/hyper-plane that “best fits’ the training samples.

- So we define the error function for our algorithm so we can minimise that error.

$$E = \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

$$E = (y - \hat{y})^T (y - \hat{y})$$

$$E = y^T y - y^T \hat{y} - \hat{y}^T y + \hat{y}^T \hat{y}$$

$$E = y^T y - y^T Xw - (Xw)^T y + (Xw)^T (Xw)$$

$$E = y^T y - y^T Xw - w^T X^T y + w^T X^T Xw$$

# Multi Dimension Linear Regression

- To determine the value of *w* that minimises the error *E*, we look for where the differential of *E* with respect to *w* is zero.
- We use the Matrix Cookbook to help with the differentiation!

$$\frac{\partial E}{\partial w} = -2X^T y + 2X^T Xw$$

$$-2X^T y + 2X^T Xw = 0$$

$$X^T Xw = X^T y$$

$$w = (X^T X)^{-1} X^T y$$

# Multi Dimension Linear Regression

- We also define a function which we can use to score how well derived line fits.
- A value of 1 indicates a perfect fit.
- A value of 0 indicates a fit that is no better than simply predicting the mean of the input y values.
- A negative value indicates a fit that is even worse than just predicting the mean of the input y values.

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y})^2}{\sum_i (y_i - \bar{y})^2}$$

# Multi Dimension Linear Regression

```
In [1]: import numpy as np
   ...:
   ...: # Create some training samples.
   ...:
   ...: # Note the bias 1s added to the start of each raw sample.
   ...: # By addign this, we allow the  weight that corresponds
   ...: # to this column in w to play the same role that the
   ...: # variable b plays in our single dimension linear
   ...: # regression function.
   ...: X = np.array([
   ...: [1, 17.9302012052, 94.5205919533],
   ...: [1, 97.1446971852, 69.5932819844],
   ...: [1, 81.7759007845, 5.73764809688] ])
   ...:
   ...: Y = np.array([ 317, 405, 180])
   ...:
```

# Multi Dimension Linear Regression

```
In [2]: # Derive the weights that give the lowest error.
   ...: w = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, Y))

In [3]: w
Out[3]: array([-6.1064527 ,  2.06343067,  3.02694598])

In [4]: # Calcualte the predicted y values.
   ...: Yhat = np.dot(X, w)

In [5]: Yhat
Out[5]: array([ 317.,  405.,  180.])

In [6]: # Calculate a score of the accuracy of our predictions.
   ...: d1 = Y - Yhat
   ...: d2 = Y - Y.mean()
   ...: rsquared = 1 - d1.dot(d1) / d2.dot(d2)
   ...:

In [7]: rsquared
Out[7]: 1.0
```

# Multi Dimension Linear Regression

- In addition to using the X matrix to represent basic features our training data, we can can also introduce additional dimensions (i.e. columns in our X matrix) that are derived from those basic feature values.

- If we introduce derived features whose values are powers of basic features, our multi-dimensional linear regression can then derive polynomial curves, planes and hyper-planes.

# Multi Dimension Linear Regression

- For example, if we have just one basic feature in each sample of X, we can include a range of powers of that value into our X matrix like this:

$$X = \begin{bmatrix} {x_1}^0 & {x_1}^1 & {x_1}^2 & {x_1}^3 \\ {x_{\dots}}^0 & {x_{\dots}}^1 & {x_{\dots}}^2 & {x_{\dots}}^3 \\ {x_n}^0 & {x_n}^1 & {x_n}^2 & {x_n}^3 \end{bmatrix}$$

- In non-matrix form our multi-dimensional linear equation is:

$$\hat{y_i} = w_o X_{io} + w_1 X_{i1} + \dots w_d X_{id}$$

- Inserting the powers of the basic feature that we have introduced this becomes a polynomial:

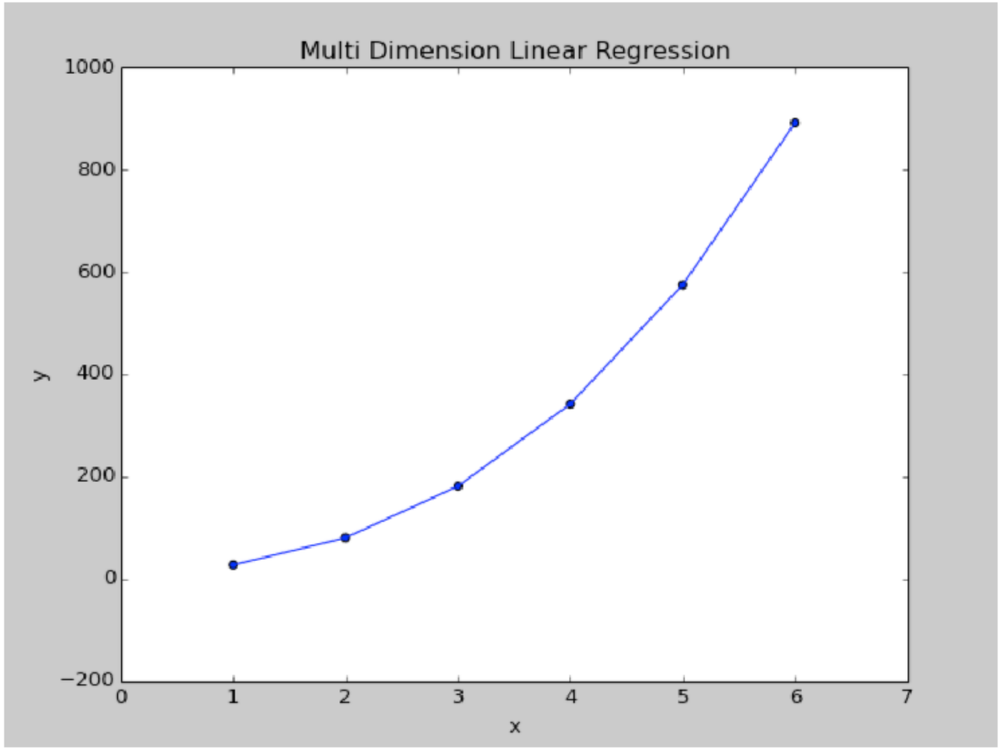
$$\hat{y_i} = w_o {X_i}^0 + w_1 {X_i}^1 + w_1 {X_i}^2 + w_1 {X_i}^3$$

# Multi Dimension Linear Regression

```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: import math
   ...:
   ...: # Create some training samples.
   ...:
   ...: Xraw = np.array([1, 2, 3, 4, 5, 6])
   ...:
   ...: # Our original input sample are 2, 3, 9 & 12. We choose
   ...: # to include each value to the power of 0, 1 & 2 in
   ...: # separate columns of our input.
   ...: #
   ...: # By preparing our input values in this way, the algorithm
   ...: # can derive a polynomial (i.e. a curved plane or hyper-plane).
   ...: X = np.array([
   ...:   [math.pow(Xraw[0], 0), math.pow(Xraw[0], 1), math.pow(Xraw[0],2), math.pow(Xraw[0],3)],
   ...:   [math.pow(Xraw[1], 0), math.pow(Xraw[1], 1), math.pow(Xraw[1],2), math.pow(Xraw[1],3)],
   ...:   [math.pow(Xraw[2], 0), math.pow(Xraw[2], 1), math.pow(Xraw[2],2), math.pow(Xraw[2],3)],
   ...:   [math.pow(Xraw[3], 0), math.pow(Xraw[3], 1), math.pow(Xraw[3],2), math.pow(Xraw[3],3)],
   ...:   [math.pow(Xraw[4], 0), math.pow(Xraw[4], 1), math.pow(Xraw[4],2), math.pow(Xraw[4],3)],
   ...:   [math.pow(Xraw[5], 0), math.pow(Xraw[5], 1), math.pow(Xraw[5],2), math.pow(Xraw[5],3)] ])
   ...:
   ...: Y = np.array([27, 80, 181, 342, 575, 892])
   ...:

In [2]: X
Out[2]:
array([[   1.,    1.,    1.,    1.],
       [   1.,    2.,    4.,    8.],
       [   1.,    3.,    9.,   27.],
       [   1.,    4.,   16.,   64.],
       [   1.,    5.,   25.,  125.],
       [   1.,    6.,   36.,  216.]])

In [3]: Y
Out[3]: array([ 27,  80, 181, 342, 575, 892])
```

# Multi Dimension Linear Regression

```
In [4]: # Derive the weights that give the lowest error.
   ...: w = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, Y))
   ...:
   ...: # Calcualte the predicted y values.
   ...: Yhat = np.dot(X, w)
   ...:
   ...: # Calculate a score of the accuracy of our predictions.
   ...: d1 = Y - Yhat
   ...: d2 = Y - Y.mean()
   ...: rsquared = 1 - d1.dot(d1) / d2.dot(d2)
   ...:
   ...: # Plot the training points and predicted line of best fit
   ...: plt.scatter(Xraw, Y)
   ...: plt.plot(Xraw, Yhat)
   ...: plt.title("Multi Dimension Linear Regression")
   ...: plt.xlabel("x")
   ...: plt.ylabel("y")
   ...: plt.show()
   ...:
```



# Gradient Descent

# Singular Matrices

- As we have seen, we can use numpy's linalg.solve() function to determine the value of the weights that result in the lowest possible error.
- But this doesn't work if np.dot(X.T, X) is a singular matrix.
- It results in the matrix equivalent of a divide by zero.
- Gradient descent is an alternative approach to determining the optimal weights that in works for all cases, including this singular matrix case.

```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...:
   ...: # Create some training samples in which np.dot(X.T, X) is a singular matrix
   ...: X = np.array([
   ...:   [1, 1, 0],
   ...:   [1, 1, 0],
   ...:   [1, 1, 0],
   ...:   [1, 1, 0],
   ...:   [1, 1, 0],
   ...:   [1, 0, 1],
   ...:   [1, 0, 1],
   ...:   [1, 0, 1],
   ...:   [1, 0, 1],
   ...:   [1, 0, 1] ])
   ...:
   ...: Y = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
   ...:
   ...: # Attempt to derive the weights that give the lowest error and
   ...: # observe that this fails because np.dot(X.T, X) is a singular matrix.
   ...: #
   ...: # This is one example where we need to use an alternative technique, such
   ...: # as gradient descent, to determine the weights that give the lowest
   ...: # error.
   ...: w = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, Y))
   ...:

LinAlgError: Singular matrix
```

# Gradient Descent

- Gradient descent is a technique we can use to find the minimum of arbitrarily complex error functions.
- In gradient descent we pick a random set of weights for our algorithm and iteratively adjust those weights in the direction of the gradient of the error with respect to each weight.
- As we iterate, the gradient approaches zero and we approach the minimum error.
- In machine learning we often use gradient descent with our error function to find the weights that give the lowest errors.

# Gradient Descent

- Here is an example with a very simple function:

$$y = x^2$$

- The gradient of this function is given by:

$$\frac{dy}{dx} = 2x$$

- We choose an random initial value for $x$ and a learning rate of 0.1 and then start descent.

- On each iteration our $x$ value is decreasing and the gradient ($2x$) is converging towards 0.

| $x$ | $\frac{dy}{dx}$ | $\frac{dy}{dx} \times 0.1$ |
|---|---|---|
| 20 | 40 | 4 |
| 16 | 32 | 3.2 |
| 12.8 | 25.6 | 2.56 |
| 10.24 | 20.48 | 2.048 |
| 8.192 | 16.384 | 1.6384 |
| 6.5536 | 13.1072 | 1.31072 |

# Gradient Descent

- The learning rate is a what is know as a hyper-parameter.
- If the learning rate is too small then convergence may take a very long time.
- If the learning rate is too large then convergence may never happen because our iterations bounce from one side of the minima to the other.
- Choosing a suitable value for hyper-parameters is an art so try different values and plot the results until you find suitable values.

# Multi Dimension Linear Regression with Gradient Descent

- For multi dimension linear regression our error function is:

$$E = (y - \hat{y})^T (y - \hat{y})$$

- Differentiating this with respect to the weights vector gives:

$$\frac{\partial E}{\partial w} = -2X^T y + 2X^T \hat{y}$$

- We can iteratively reduce the error by adjusting the weights in the direction of these gradients.

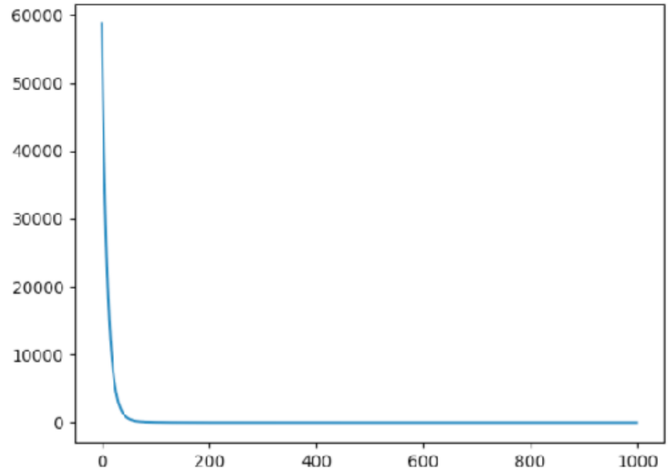
$$\frac{\partial E}{\partial w} = 2X^T (\hat{y} - y)$$

# Multi Dimension Linear Regression with Gradient Descent

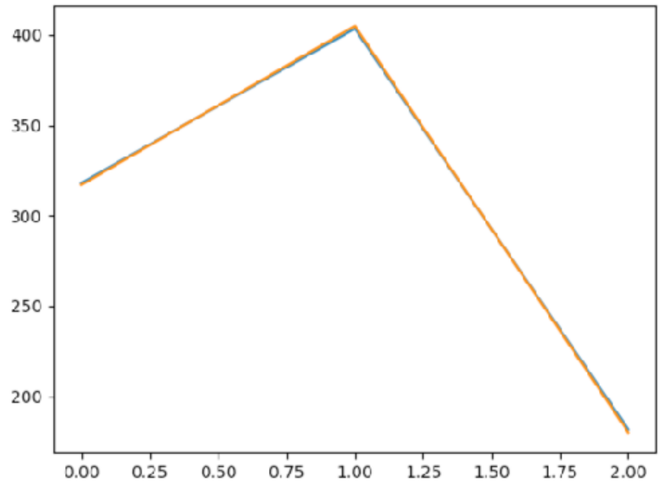
```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...:
   ...: X = np.array([
   ...: [1, 17.9302012052, 94.5205919533],
   ...: [1, 97.1446971852, 69.5932819844],
   ...: [1, 81.7759007845, 5.73764809688] ])
   ...:
   ...: Y = np.array([ 317, 405, 180])
   ...:
   ...: # Set weights to random values and ensure
   ...: # they have a variance of 1/D which is
   ...: # optimal but not required for
   ...: # descent.
   ...: w = np.random.randn(3) / np.sqrt(3)
   ...:
   ...: learning_rate = 0.000001
   ...:
   ...: # We'll store the mean squared error between Y and Yhat so we can show
   ...: # it decreases as we descend the gradient.
   ...: errors = []
   ...:
   ...: for t in range(1000):
   ...:   YHat = X.dot(w)
   ...:   delta = YHat - Y
   ...:   gradient = 2 * X.T.dot(delta)
   ...:   w = w - (learning_rate * gradient)
   ...:   error = delta.dot(delta) / 3
   ...:   errors.append(error)
   ...:
```

# Multi Dimension Linear Regression with Gradient Descent

```
In [2]: # Plot the mean squared error reducing over the 1000 iterations.
   ...: plt.plot(errors)
   ...: plt.show()
   ...:
```




```
In [3]: # Plot the predicted and actual values of Y
   ...: plt.plot(YHat, label='prediction')
   ...: plt.plot(Y, label='targets')
   ...: plt.show()
   ...:
```



# Generalisation, Over-fitting & Regularisation

# Generalisation & Over-fitting

- As we train our model with more and more data the it may start to fit the training data more and more accurately, but become worse at handling test data that we feed to it later.
- This is know as “over-fitting” and results in an increased generalisation error.
- To minimise the generalisation error we should
  - Collect as much sample data as possible.
  - Use a random subset of our sample data for training.
  - Use the remaining sample data to test how well our model copes with data it was not trained with.
- Also, experiment with adding higher degrees of polynomials ($X^2$, $X^3$, etc) as this can reduce overfitting.

# L1 Regularisation (Lasso)

- Having a large number of samples (n) with respect to the number of dimensionality (d) increases the quality of our model.
- One way to reduce the effective number of dimensions is to use those that most contribute to the signal and ignore those that mostly act as noise.
- L1 regularisation achieves this by adding a penalty that results in the weight for the dimensions that act as noise becoming 0.
- L1 regularisation encourages a sparse vector of weights in which few are non-zero and many are zero.

# L1 Regularisation (Lasso)

- In L1 regularisation we add a penalty to the error function:

$$E = (y - Xw)^T(y - Xw) + \lambda|w|$$

- Expanding this we get:

$$E = y^T y - 2y^T Xw + W^T X^T Xw + \lambda|w|$$

- Take the derivative with respect to w to find our gradient:

$$\frac{\partial E}{\partial w} = -2X^T y + 2X^T Xw + \lambda sign(w)$$

- Where *sign(w)* is *-1* if $w < 0$, *0* if $w = 0$ and *+1* if $w > 0$

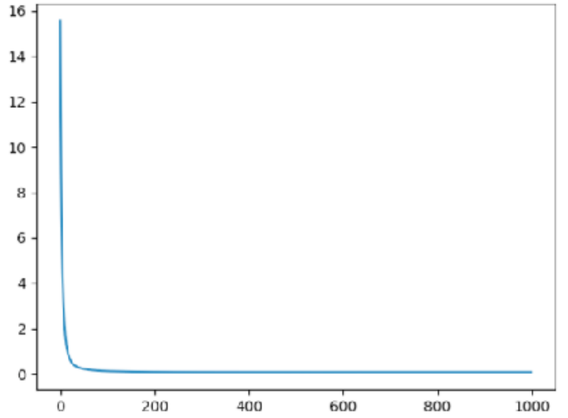
- Note that because *sign(w)* has no inverse function we cannot solve for *w* and so must use gradient descent.

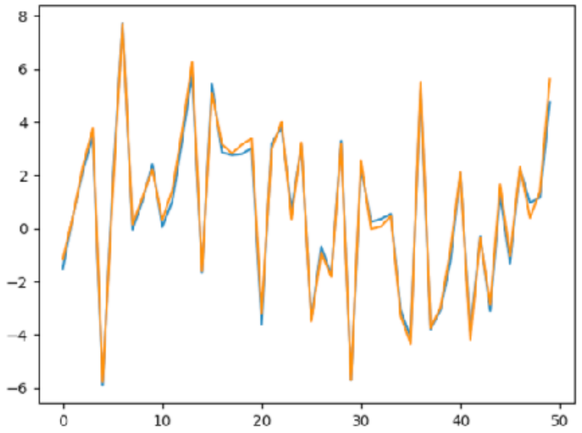
# L1 Regularisation (Lasso)

```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...:
   ...: # Create some training samples
   ...:
   ...: # To demonstrate L1 regularisation we fabricate training data
   ...: # with 50 dimensions in our X matrix, where only 3 of which
   ...: # contribute significantly to the values in our Y vector.
   ...:
   ...: # Construct X
   ...: X = (np.random.random((50,50)) - 0.5) * 10
   ...:
   ...: # Construct Y
   ...: actualW = np.array([1, 0.5, -0.5] + [0]*47)
   ...: Y = X.dot(actualW) + np.random.randn(50) * 0.5
   ...:
   ...: # Use gradient descent with L1 regularisation to derive the
   ...: # wieghts from the training samples.
   ...: w = np.random.randn(50) / np.sqrt(50)
   ...: learning_rate = 0.0001
   ...: errors = []
   ...:
   ...: for t in range(1000):
   ...:   YHat = X.dot(w)
   ...:   delta = YHat - Y
   ...:   gradient = 2 * X.T.dot(delta)
   ...:   l1 = 10 * np.sign(w);
   ...:   w = w - (learning_rate * (gradient + l1))
   ...:   error = delta.dot(delta) / 50
   ...:   errors.append(error)
   ...:
```

# L1 Regularisation (Lasso)

```
In [2]: # Plot the mean squared error reducing over the 1000 iterations.
   ...: plt.plot(errors)
   ...: plt.show()
   ...:

In [3]: # Plot the predicted and actual values of Y
   ...: plt.plot(YHat, label='prediction')
   ...: plt.plot(Y, label='targets')
   ...: plt.show()
   ...:

In [4]: # Note how all but the first three derived weights are very
   ...: # close to zero.
   ...: print(w)
   ...:
[  9.83809630e-01   5.24309389e-01  -4.46045446e-01   4.21896156e-04
   1.72398079e-03   8.44891535e-04   6.68148640e-04   7.49178554e-03
   1.90923632e-02   1.28152152e-04   1.08966884e-03   8.13793034e-04
   3.59965572e-04  -8.32167622e-05   5.07235004e-05  -8.97555390e-05
   1.01142049e-03  -3.75627603e-03   2.62711012e-03   1.34381551e-03
   1.05314389e-04  -9.35410936e-06  -9.45443120e-03  -1.68105408e-02
  -4.25096403e-04   1.04176693e-02   1.18205511e-03   5.69307315e-04
   2.64839100e-04   1.41236451e-03   5.74420504e-04  -6.25544808e-04
   2.47675797e-03   7.05679324e-03  -7.39973615e-03  -1.11315488e-03
   1.87274080e-04  -1.92970861e-02  -2.69695615e-02   3.18597952e-04
   5.47769457e-04   8.63886380e-04  -6.10865339e-02  -2.86751922e-05
  -1.67620559e-02  -1.50272746e-03  -4.98823235e-04  -6.50550234e-04
  -9.06626486e-04   1.58420645e-02]
```



# L2 Regularisation (Ridge)

- Another way to reduce the complexity of our model and prevent overfitting to outliers is L2 regression, which is also known as ridge regression.

- In L2 Regularisation we introduce an additional term to the cost function that has the effect of penalising large weights and thereby minimising this skew.

# L2 Regularisation (Ridge)

- In L2 regularisation we the sum of the squares of the weights to the error function.

$$E = (y - Xw)^T (y - Xw) + \lambda w^T w$$

- Expanding this we get:

$$E = y^T y - 2y^T Xw + W^T X^T Xw + \lambda w^T w$$

- Take the derivative with respect to w to find our gradient:

$$\frac{\partial E}{\partial w} = -2X^T y + 2X^T Xw + 2\lambda w$$

# L2 Regularisation (Ridge)

- Solving for the values of w that give minimal error:

$$-2X^T y + 2X^T Xw + 2\lambda w = 0$$

$$(\lambda I + X^T X)w = X^T y$$

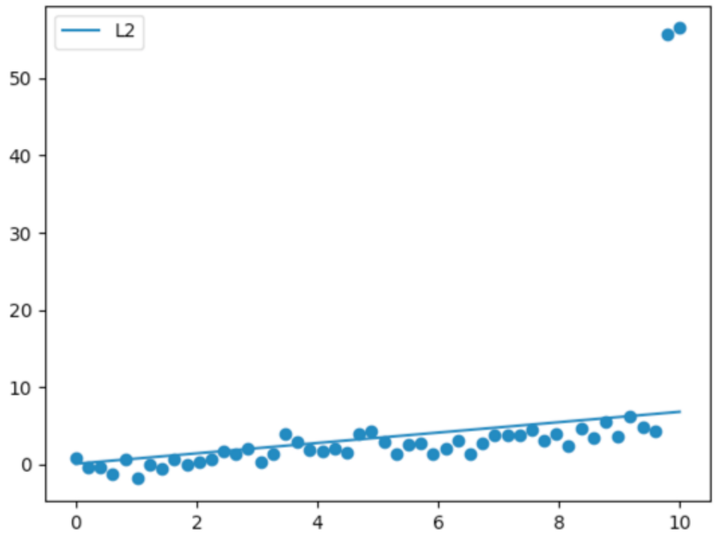
$$w = (\lambda I + X^T X)^{-1} X^T y$$

# L2 Regularisation (Ridge)

```
In [1]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...:
   ...: # Create some training samples
   ...:
   ...: # To demonstrate L2 regularisations ability to minimize the impact
   ...: # of outliers, we fabricate training data where X and Y are roughly
   ...: # linearly related and then insert a few outliers into Y.
   ...:
   ...: # Construct X
   ...: X = np.linspace(0, 10, 50)
   ...:
   ...: # Construct Y
   ...: Y = 0.5*X + np.random.randn(50)
   ...: Y[-1] += 50
   ...: Y[-2] += 50
   ...:
   ...: # Add a bias column to X.
   ...: X = np.vstack([np.ones(50), X]).T
   ...:
   ...: # Calculate the weights that give mimumum error using L2 regularisation
   ...: # and the corresponding predicated Y values. Use a L2 hyper-parameter
   ...: # value of 1000.
   ...: l2 = 1000
   ...: w_with_l2 = np.linalg.solve(l2*np.eye(2) + X.T.dot(X), X.T.dot(Y))
   ...: YHat_with_l2 = X.dot(w_with_l2)
   ...:
```

# L2 Regularisation (Ridge)

```
In [2]: # Plot the training data and solutions.
   ...: plt.scatter(X[:,1], Y)
   ...: plt.plot(X[:,1], YHat_with_l2, label="L2")
   ...: plt.legend()
   ...: plt.show()
   ...:
```



# L1 & L2 Regularisation (Elastic Net)

- L1 Regularisation minimises the impact of dimensions that have low weights and are thus largely “noise”.
- L2 Regularisation minimise the impacts of outliers in our training data.
- L1 & L2 Regularisation can be used together and the combination is referred to as Elastic Net regularisation.
- Because the differential of the error function contains the sigmoid which has no inverse, we cannot solve for *w* and must use gradient descent.

# Categorical Inputs

# One-hot Encoding

- When some inputs are categories (e.g. gender) rather than numbers (e.g. age) we need to represent the category values as numbers so they can be used in our linear regression equations.

- In *one-hot encoding* we allocate each category value it's own dimension in the inputs. So, for example, we allocate $X_1$ to Audi, $X_2$ to BMW & $X_3$ to Mercedes.

  - For Audi X = [1,0,0]

  - For BMW X = [0,1,0])

  - For Mercedes X = [0,0,1]

# Summary

- Single Dimension Linear Regression
- Multi Dimension Linear Regression
- Gradient Descent
- Generalisation, Over-fitting & Regularisation
- Categorical Inputs