

Machine Learning



Machine Learning

Lecture: Linear and Multivariate Regression

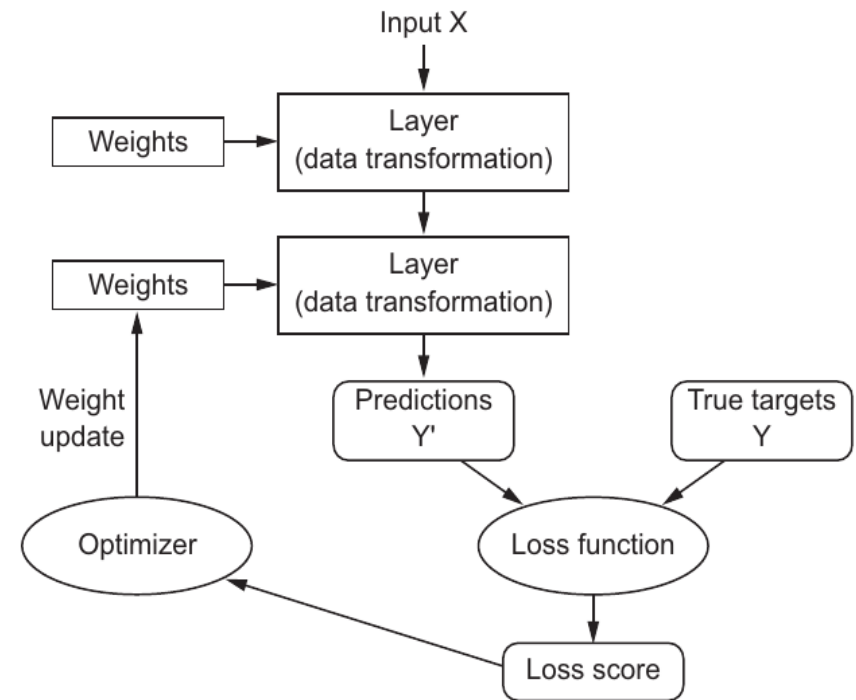
Ted Scully

Recap of MLR Methodology

f_1	f_2	f_3	f_4	...	f_n	Y
x_1^1	x_2^1	x_3^1	x_4^1	..	x_n^1	y^1
x_1^2	x_2^2	x_3^2	x_4^2	..	x_n^2	y^2
x_1^3	x_2^3	x_3^3	x_4^3	..	x_n^3	y^3
..

$$h(x) = \lambda_1 x_1 + \dots + \lambda_n x_n + b$$

Gradient Descent Optimizer



$$\frac{1}{m+2} \sum_{i=0}^m ((h(x^i) - y^i))^2$$

Multiple (Multi-variate) Linear Regression

- As we did with linear regression we can use gradient descent to find the optimal values for each λ_j and b
- **Repeat {**
$$\lambda_j = \lambda_j - \alpha \frac{\partial}{\partial \lambda_j} C(W, b)$$
$$b = b - \alpha \frac{\partial}{\partial b} C(W, b)$$
}

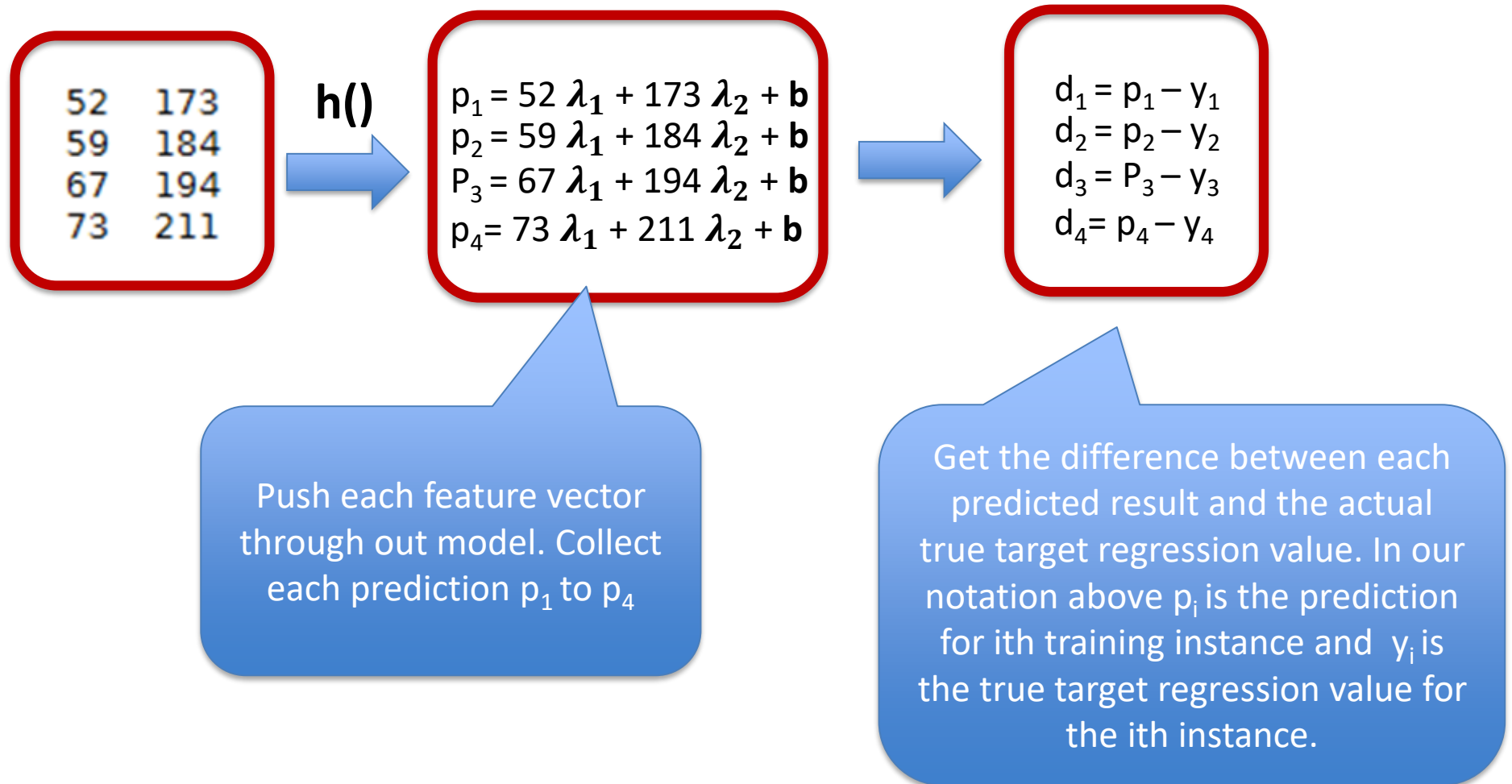
Where:

$$\frac{\partial}{\partial \lambda_j} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_j^i)$$

$$\frac{\partial}{\partial b} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))$$

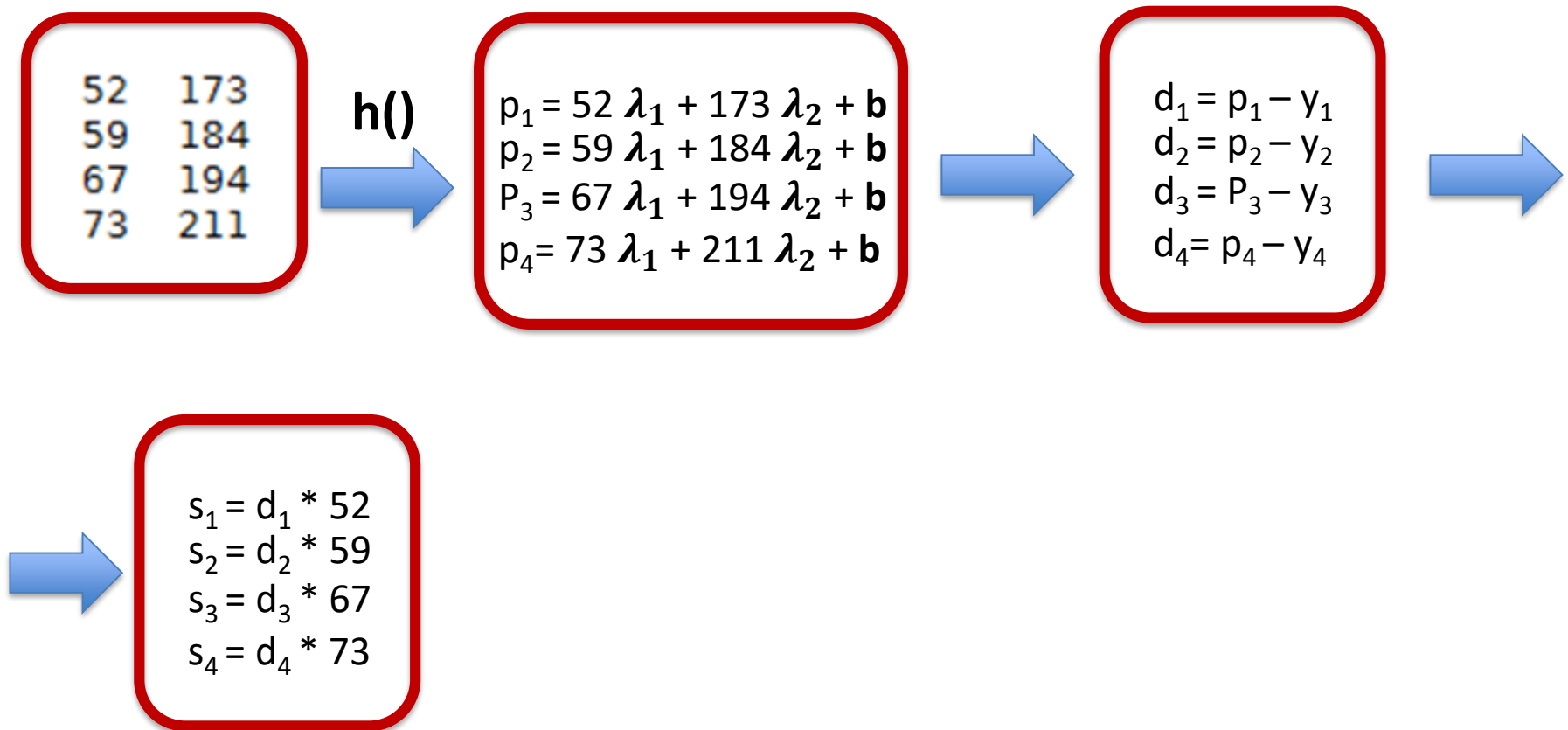
- So in this example lets assume that we want to update λ_1

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(\underline{x_1^i})$$



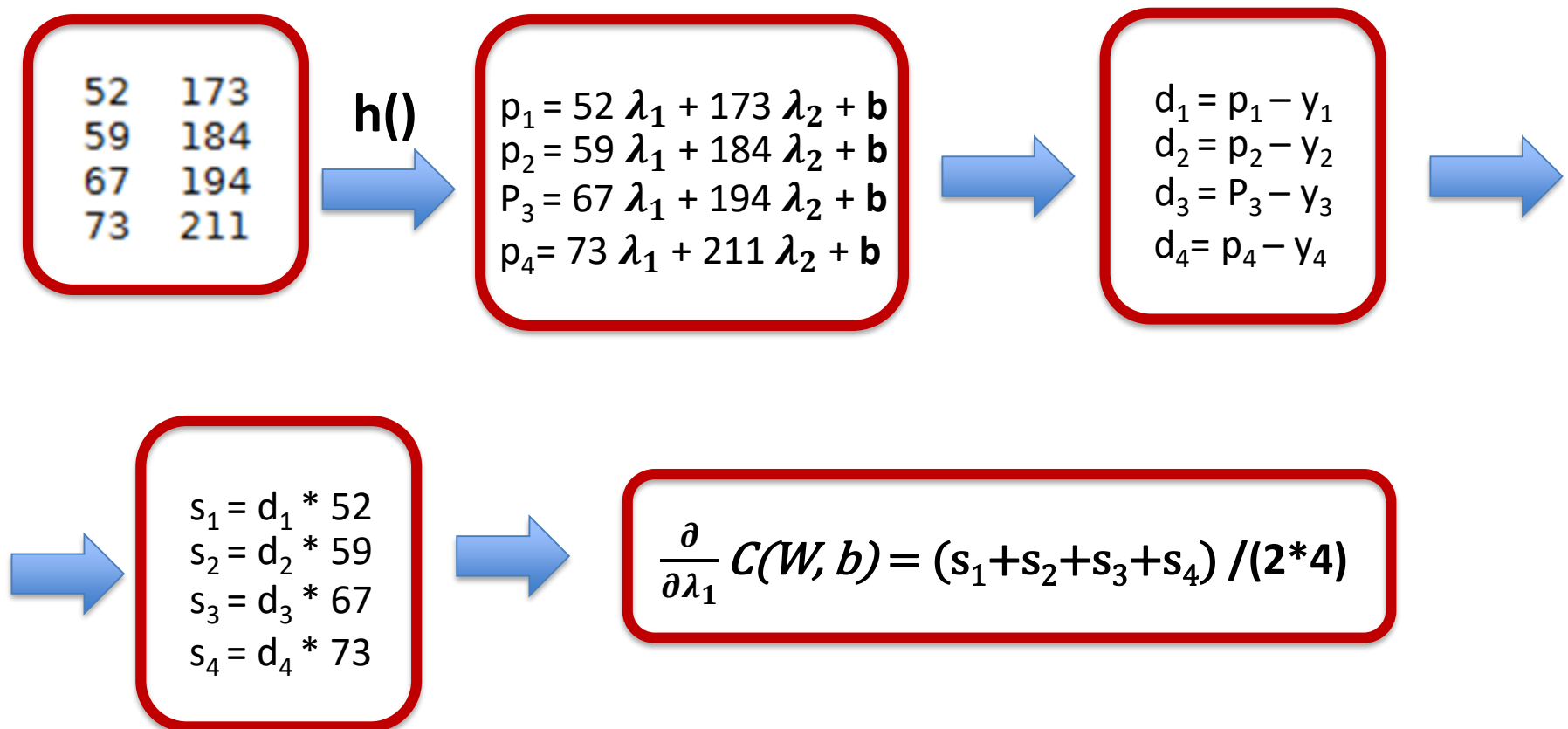
- So in this example lets assume that we want to update λ_1
- Therefore, below we calculate the partial derivitave for λ_1
- Remember m is the number of training instances therefore m = 4 below

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$



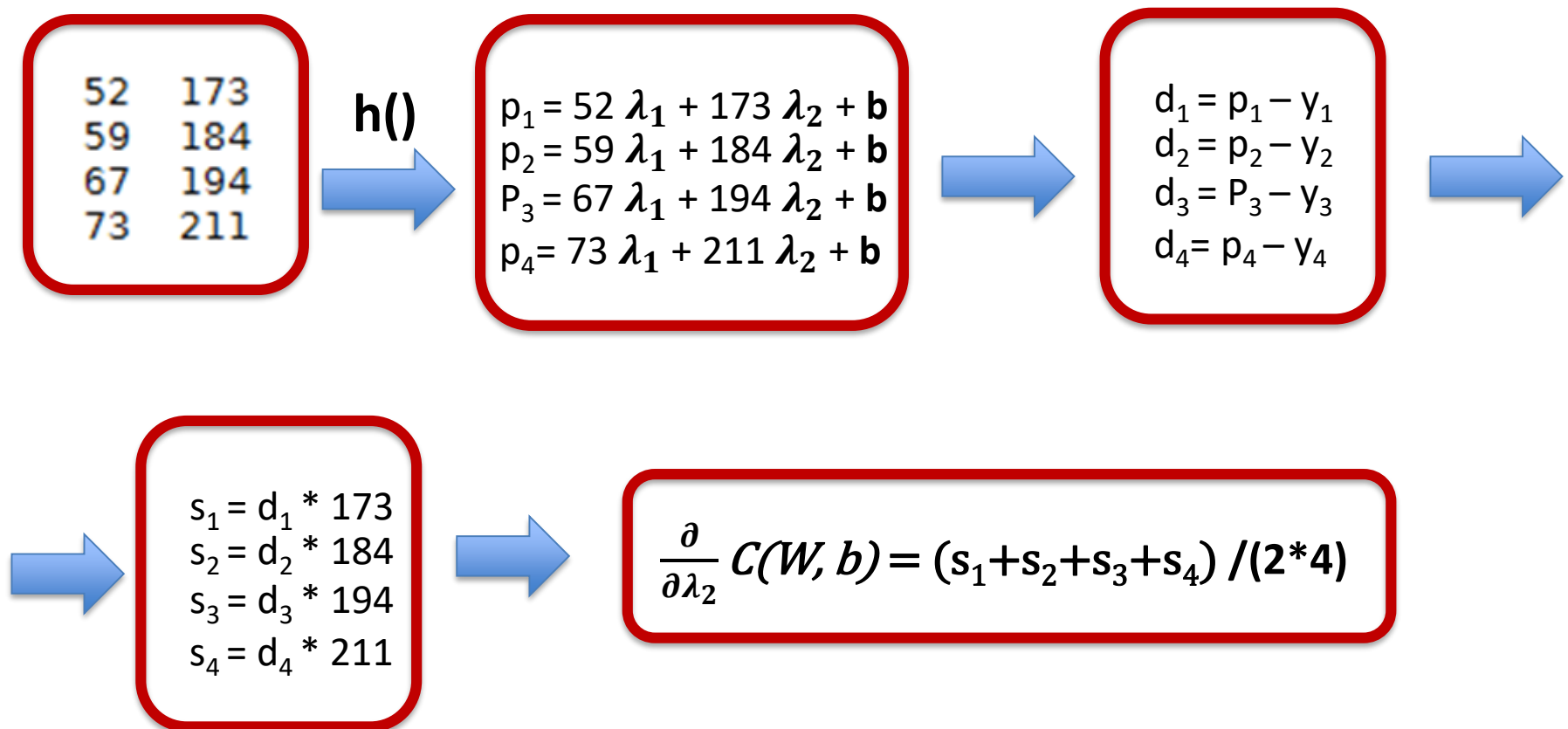
- So in this example lets assume that we want to update λ_1
- Therefore, below we calculate the partial derivitave for λ_1
- Remember m is the number of training instances therefore m = 4 below

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$



- So in this example lets assume that we want to update λ_2
- Therefore, below we calculate the partial derivitave for λ_2
- Remember m is the number of training instances therefore m = 4 below

$$\frac{\partial}{\partial \lambda_2} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_2^i)$$



Contents

1. Introduction to Linear Regression
2. Applying Gradient Decent to Linear Regression
3. Assessing Performance for Regression Models
4. Multi-Variate Linear Regression
5. Review of Matrices and Broadcasting in Python
6. Logistic Regression
7. Vectorized Logistic Regression
8. Neural Networks

Classification

- ▶ So far we have only looked at performing regression analysis (that is building a model to predict a numerical value). However, we are now going to consider the problem of **binary classification**. That is, trying to predict which of two classes a specific data instance belongs to.
- ▶ We will use a power station dataset*.
- ▶ The dataset contains measurements of the revolutions per minute (**RPM**) that power stations generators are running at, the amount of **vibration** (VIBRATION) in the generators and an indicator to show whether the generators proved to be working or faulty the day after these measurements were taken.
- ▶ The objective is to build a model that would help identify upcoming generator failures before they happen, which could improve safety and save money on maintenance.

* Power station dataset comes from Fundamentals of Machine Learning for Predictive Data Analytics – Kelleher et al.

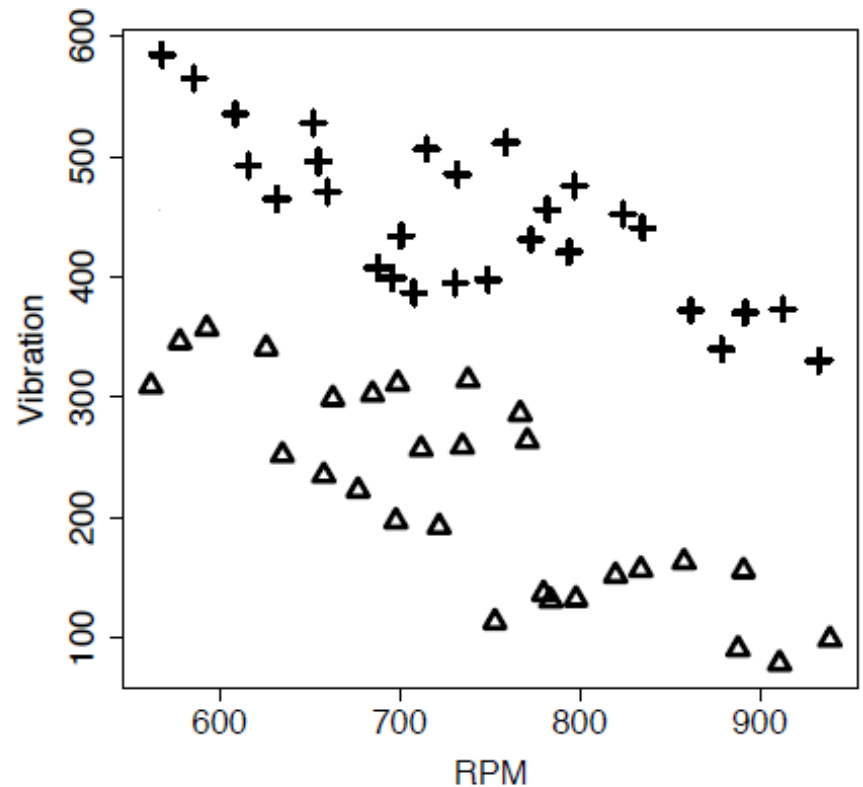
ID	RPM	VIBRATION	STATUS
1	568	585	good
2	586	565	good
3	609	536	good
4	616	492	good
5	632	465	good
6	652	528	good
7	655	496	good
8	660	471	good
9	688	408	good
10	696	399	good
11	708	387	good
12	701	434	good
13	715	506	good
14	732	485	good
15	731	395	good
16	749	398	good
17	759	512	good
18	773	431	good
19	782	456	good
20	797	476	good
21	794	421	good
22	824	452	good
23	835	441	good
24	862	372	good
25	879	340	good
26	892	370	good
27	913	373	good
28	933	330	good

ID	RPM	VIBRATION	STATUS
29	562	309	faulty
30	578	346	faulty
31	593	357	faulty
32	626	341	faulty
33	635	252	faulty
34	658	235	faulty
35	663	299	faulty
36	677	223	faulty
37	685	303	faulty
38	698	197	faulty
39	699	311	faulty
40	712	257	faulty
41	722	193	faulty
42	735	259	faulty
43	738	314	faulty
44	753	113	faulty
45	767	286	faulty
46	771	264	faulty
47	780	137	faulty
48	784	131	faulty
49	798	132	faulty
50	820	152	faulty
51	834	157	faulty
52	858	163	faulty
53	888	91	faulty
54	891	156	faulty
55	911	79	faulty
56	939	99	faulty

- ▶ You will notice in this problem we are not trying to predict a numerical value but instead we are predicting a categorical value (either good or faulty).

Classification

- ▶ The scatter plot shows the relationship between the RPM and VIBRATION feature value of the each feature.
- ▶ The **good instances** are show as crosses and the **faulty** as triangles.
- ▶ We are going to consider the problem of binary classification. That is trying to predict which of the two classes a specific data instance belongs to.
- ▶ We can see from the scatter plot that we can separate both classes by drawing a straight line.



Classification

- ▶ This depicted line is referred to as a **decision boundary**.

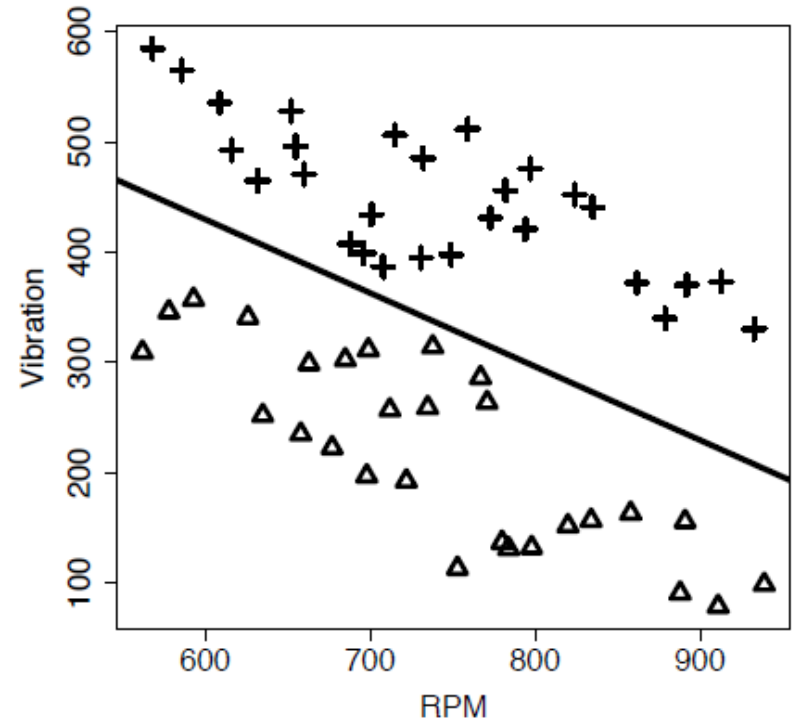
- ▶ Because this line perfectly separates the two classes the data is said to be **linearly separable** given the features.

- ▶ The decision boundary on the right is just a straight line and can be defined as a line as follows:

- ▶ $VIBRATION = (-0.667 * RPM) + 830$ or

- ▶ $830 - (0.667 * RPM) - VIBRATION = 0$

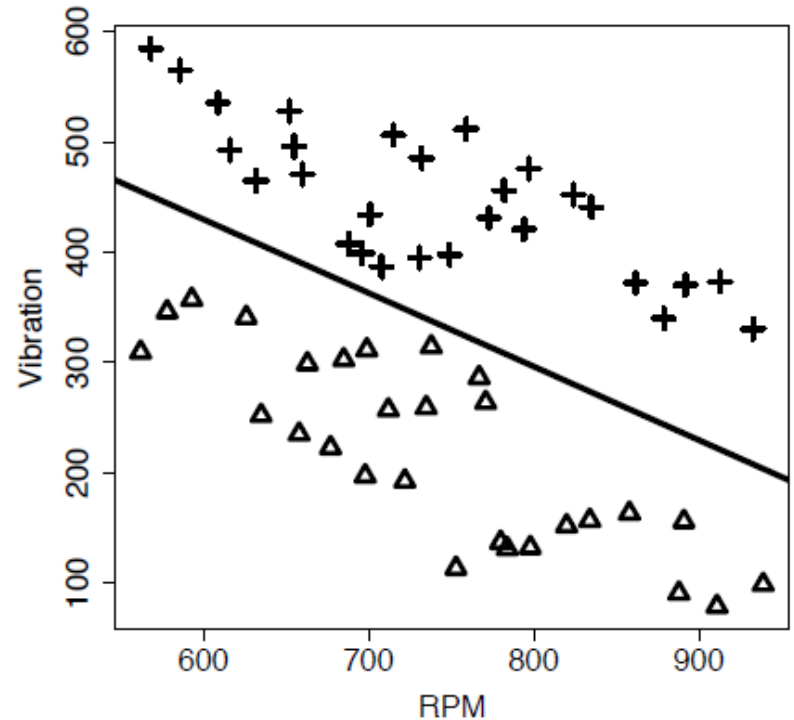
- ▶ What is interesting about the decision boundary is that it **allows us to differentiate between the two classes**.



ID	RPM	VIBRATION	STATUS	ID	RPM	VIBRATION	STATUS
1	568	585	good	29	562	309	faulty
2	586	565	good	30	578	346	faulty
3	609	536	good	31	593	357	faulty
4	616	492	good	32	626	341	faulty
5	632	465	good	33	635	252	faulty
6	652	528	good	34	658	235	faulty
7	655	496	good	35	663	299	faulty
8	660	471	good	36	677	223	faulty
9	688	408	good	37	685	303	faulty
10	696	399	good	38	698	197	faulty
11	708	387	good	39	699	311	faulty
12	701	434	good	40	712	257	faulty
13	715	506	good	41	722	193	faulty
14	732	485	good	42	735	259	faulty
15	731	395	good	43	738	314	faulty
16	749	398	good	44	753	113	faulty
17	759	512	good	45	767	286	faulty
18	773	431	good	46	771	264	faulty
19	782	456	good	47	780	137	faulty
20	797	476	good	48	784	131	faulty
21	794	421	good	49	798	132	faulty
22	824	452	good	50	820	152	faulty
23	835	441	good	51	834	157	faulty
24	862	372	good	52	858	163	faulty
25	879	340	good	53	888	91	faulty
26	892	370	good	54	891	156	faulty
27	913	373	good	55	911	79	faulty
28	933	330	good	56	939	99	faulty

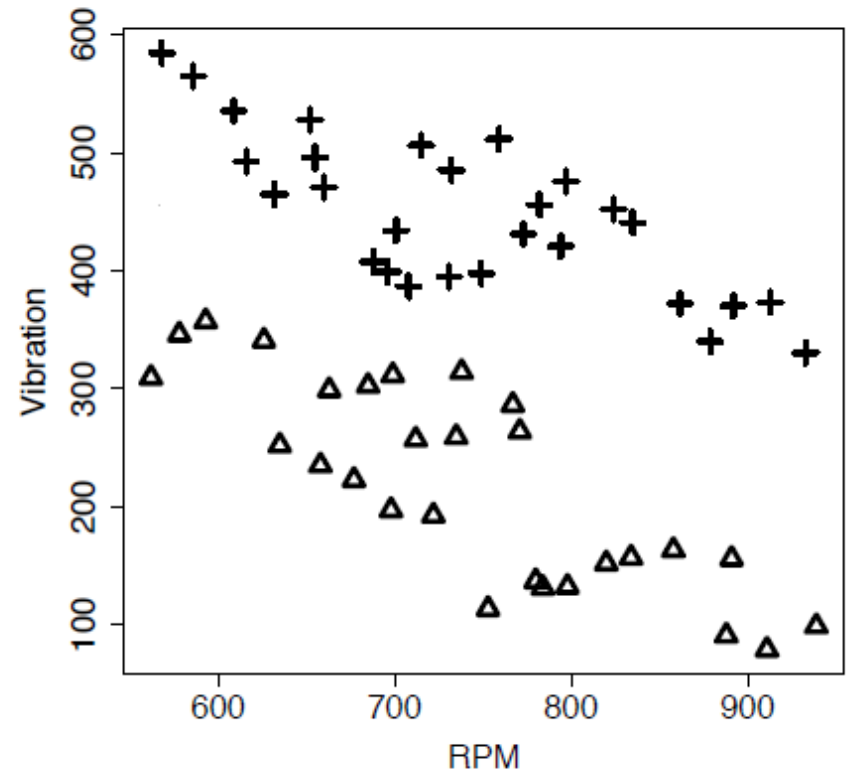
Classification

- ▶ More specifically the feature values for all instances below the decision boundary will result in a positive value, while all instances above the boundary will result in a negative value.
- ▶ For example, taking the instance **(933, 330)**, classified as **good** in our training data and plugging it into the equation above we get
- ▶ $830 - (0.667 * 933) - (1 * 330) = -122.311$
- ▶ Taking the instance **(911, 79)** classified as **bad** in our training data we get
- ▶ $830 - (0.667 * 911) - (1 * 79) = 143.363$
- ▶ What's also interesting is that the further we get away from the decision boundary the larger the positive or negative value outputted.



Classification

- ▶ So here is our idea. Maybe we could start off with a **random guess** for the equation of a line (more specifically a random guess for the weights/coefficients) that will represent our decision boundary (in a similar way that we started off with a random guess for the weights in linear regression).
- ▶ When we start off it's probably going to give poor results.
- ▶ But maybe we can **adjust the weights** that define this line so that it continually gets better and better (in the same way that we did with linear regression)

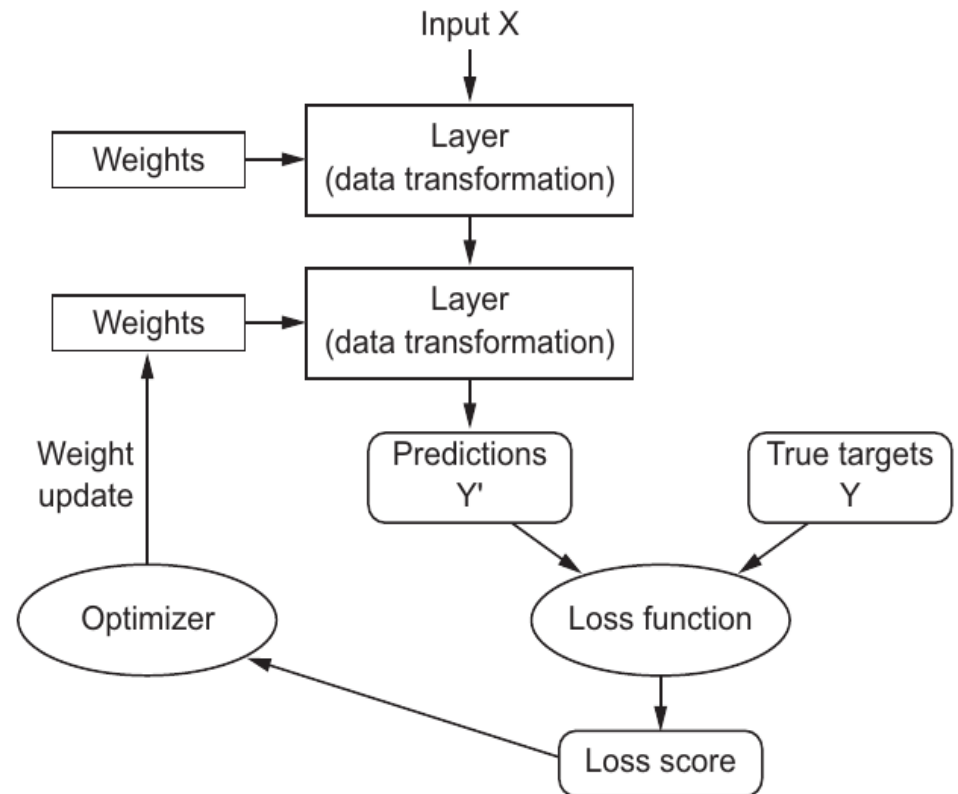


So let's consider our model first.

This is a binary classification problem. Therefore, we really want our model to be making a prediction that is 0 or 1 ... or a numerical value within this range.

So just using the same model as we used in linear regression doesn't make a lot of sense because it will output positive or negative numerical values not within the range 0 - 1.

$$\lambda_1 x_1 + \dots + \lambda_n x_n + b$$



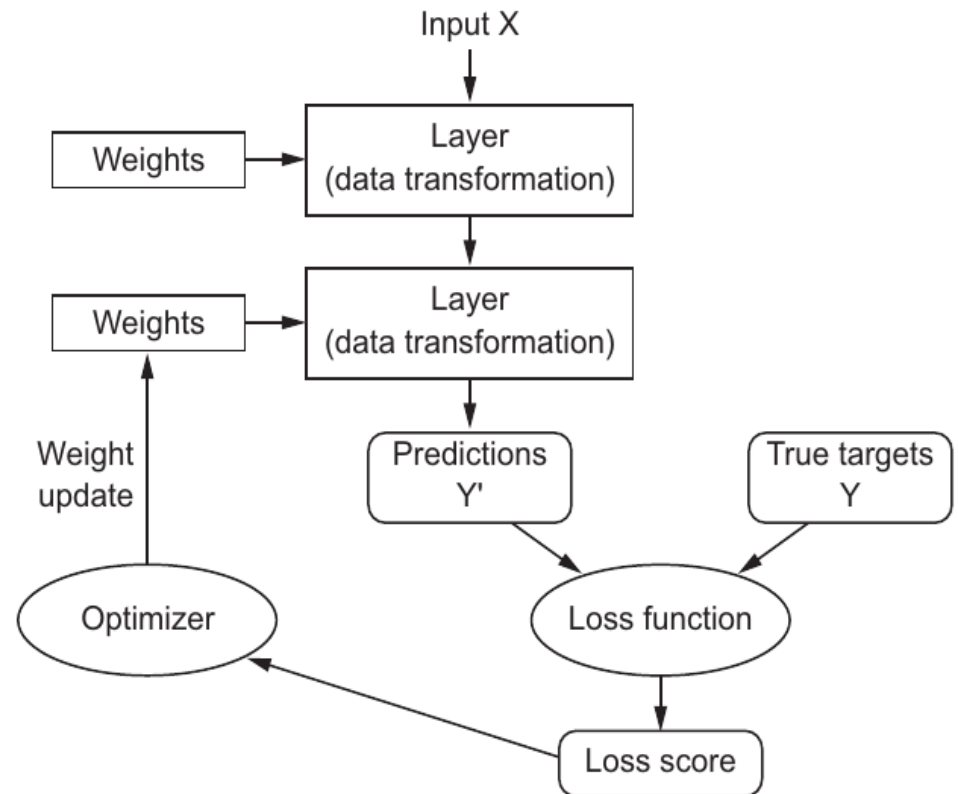
So let's consider our model first.

This is a binary classification problem. Therefore, we really want our model to be making a prediction that is 0 or 1 ... or a numerical value within this range.

So just using the same model as we used in linear regression doesn't make a lot of sense because it will output positive or negative numerical values not within the range 0 - 1.

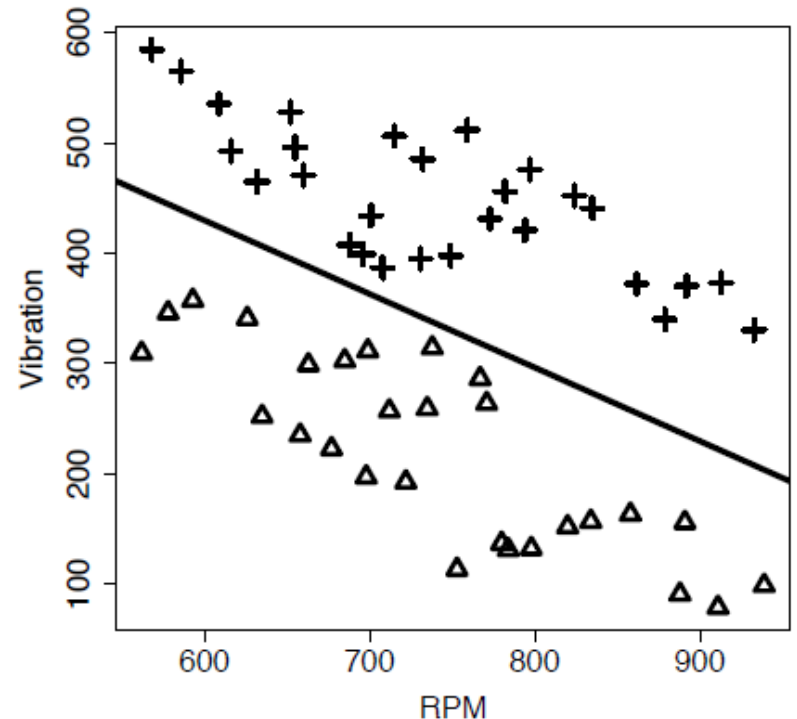
Maybe we can modify or augment our hypothesis in such a way that we can make it output a predicted value between 0 and 1.

$$\lambda_1 x_1 + \dots + \lambda_n x_n + b$$



Classification

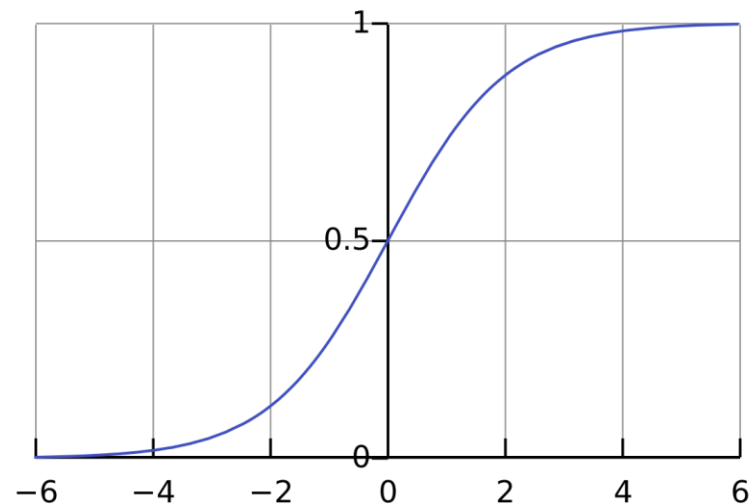
- ▶ Maybe we could use this **negative/positive property** of the equation of a line to build our model. Remember all instances **below** the line will result in a **positive value**, while all instances **above** the boundary will result in a **negative value**.
- ▶ Therefore, if the line produces a negative output then we could output a value of 0, if the line produces a positive output then we output a 1.
- ▶ What would be even better is if we could return a **value between 0 and 1**. The closer the value is to 1 the more confident we are that it belongs to class 1 and vica versa (the closer the predicted value is to 0 then the more confident we are that it is of class 0.)

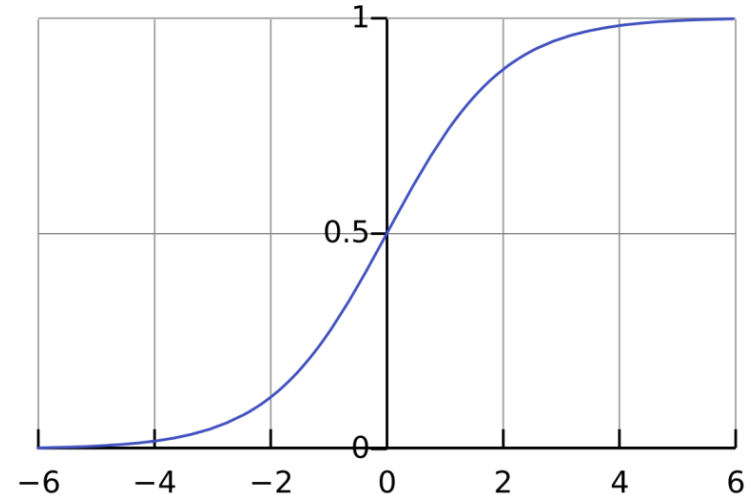
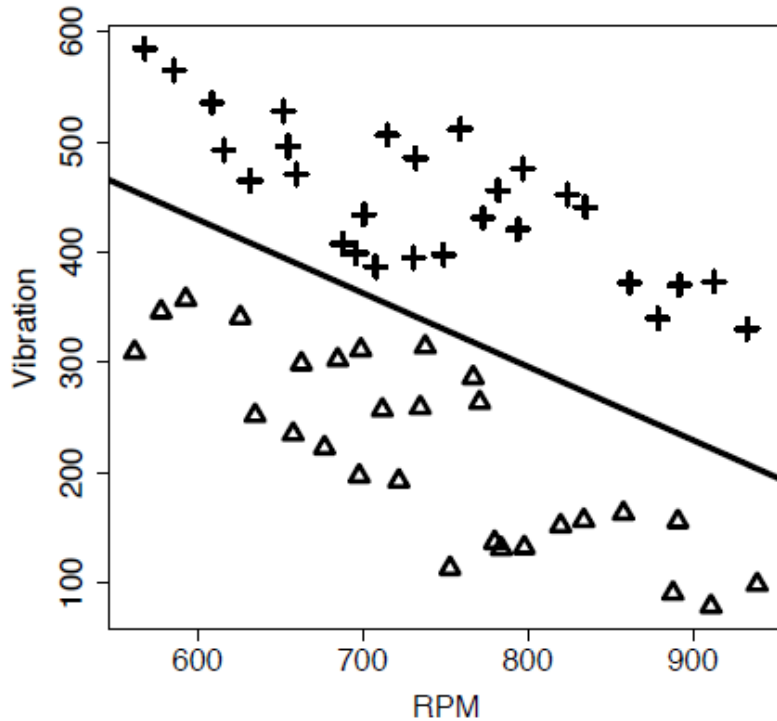


Sigmoid (Logistic) Activation Function

- ▶ For the next few slide we will represent our equation of the line as $X\lambda^T + b$
- ▶ So even though our equation of the line function ($X\lambda^T + b$) can potentially output large negative or positive values, we can use the sign of the outputted value (positive or negative) to predict the categorical target feature.
- ▶ A function that we can use to achieve this is the **Sigmoid (Logisitc)** activation function.
- ▶ Notice the output of the logistic function as always between 0 and 1. If the value inputted is negative then the output will be less than 0.5. If the value inputted to the function is positive then the value outputted will be greater than 0.5.

$$\text{logistic}(x) = \frac{1}{1+e^{-x}}$$





$$\text{logistic}(X\lambda^T + b) = \frac{1}{1 + e^{-X\lambda^T + b}}$$

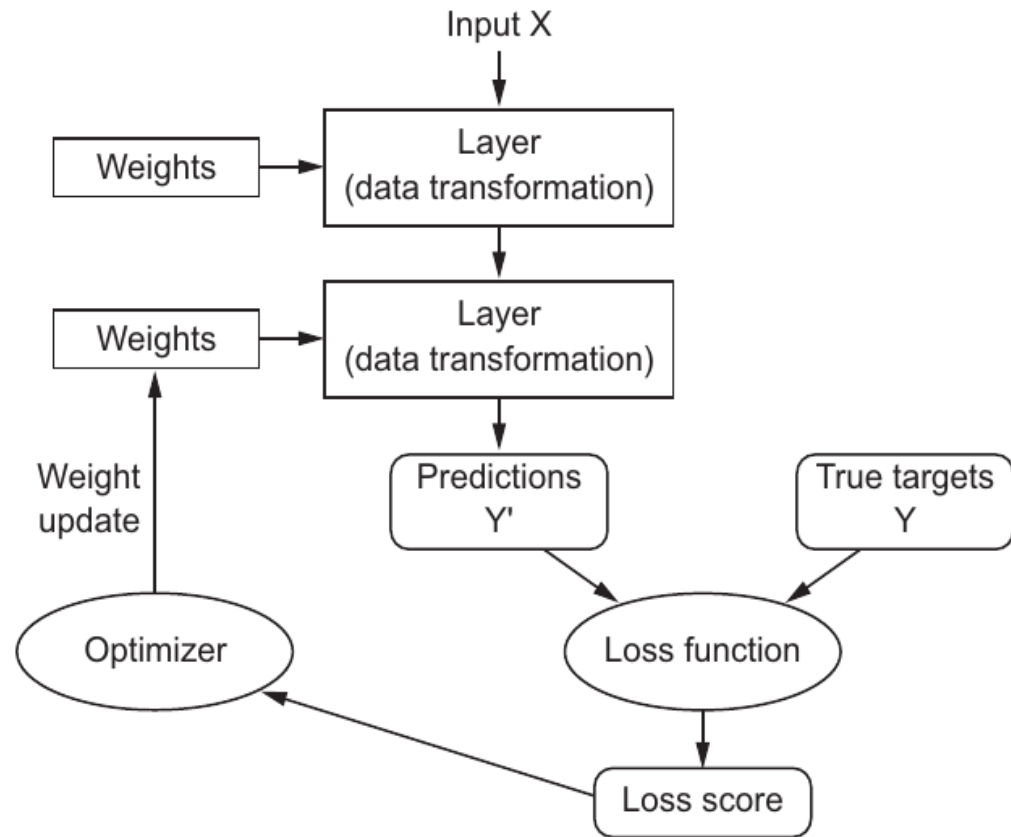
- ▶ The characteristic of this function is very useful for our classification problem as it always outputs a value between 1 and 0.
- ▶ If $X\lambda^T + b > 0$ then the logistic function returns a value greater than 0.5.
- ▶ If $X\lambda^T + b < 0$ then the logistic function returns a value less than 0.5

So now our model produces values in the range 0 to 1 and it is parametrized by weights.

So next we have to decide what function to use as our loss function.

Well perhaps we could use the modified MSE loss function again?


$$\text{logistic}(X\lambda^T + b) = \frac{1}{1 + e^{-X\lambda^T + b}}$$



Logistic (Sigmoid Function)

- ▶ Now that we have a way of formulating our hypothesis using the sigmoid function we can now return to our loss function (lets first consider using the same version we used for linear regression).

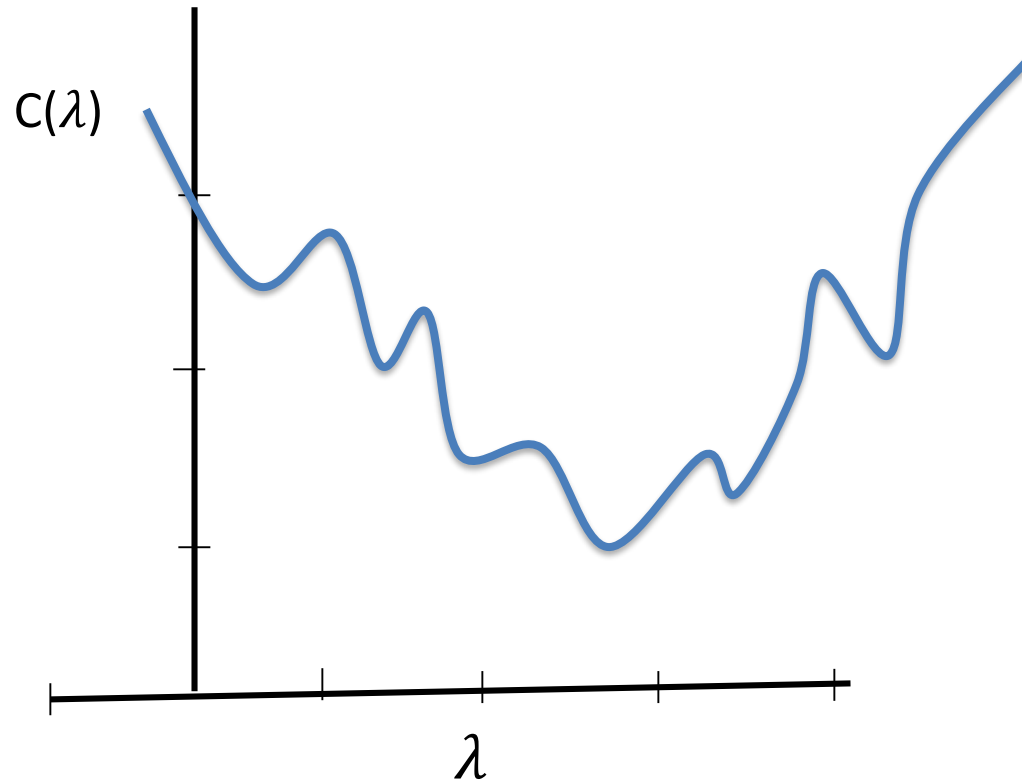
$$h(X) = \frac{1}{1 + e^{-X\lambda^T + b}}$$

$$L(W) = \frac{1}{n+2} \sum_{i=0}^n ((h(x^i) - y^i))^2$$


Logistic (Sigmoid Function)

- ▶ The problem we encounter is that if we use the existing loss function with the new sigmoid based hypothesis, it produces a non-convex function as shown below.

$$L(W) = \frac{1}{n+2} \sum_{i=0}^n ((h(x^i) - y^i))^2$$

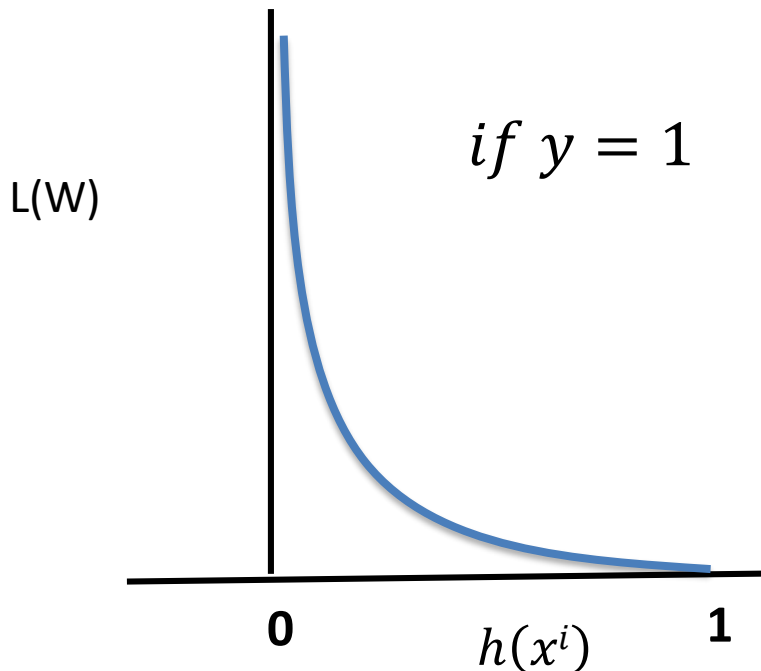


Why does this represent a problem when using gradient descent?

Cross Entropy Loss Function

- ▶ Therefore, for logistic regression we use a different loss function known as the cross entropy error function (note here I use the notion of a loss function to differentiate from the cost function in later slides):

$$L(W) = \begin{cases} -\log h(x^i) & \text{if } y = 1 \\ -\log(1 - h(x^i)) & \text{if } y = 0 \end{cases}$$



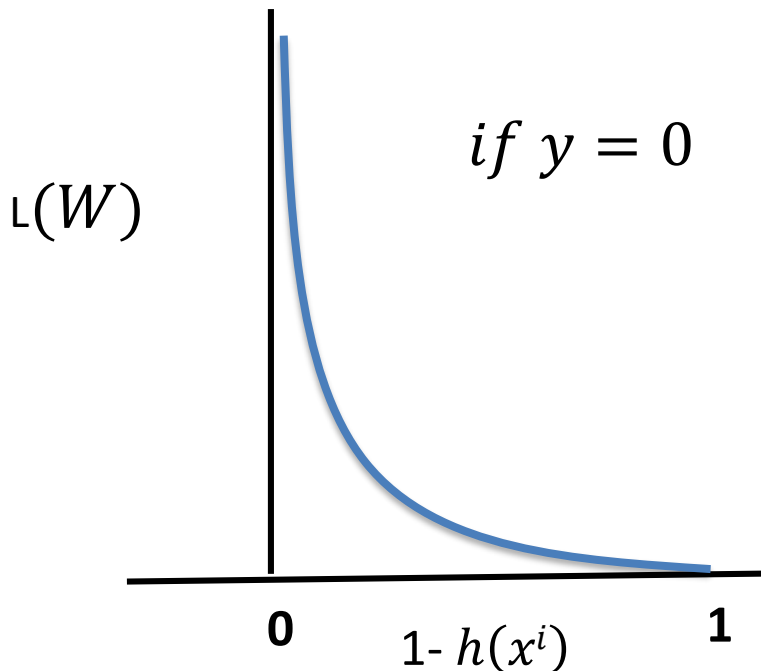
Notice if $y = 1$ (true label = 1) and our hypothesis predicts :

- A value close to 1 then our loss is very small.
- However, if $y = 1$ and our hypothesis predicts a value close to 0 then our cost will be very high.

Cross Entropy Loss Function

- Therefore, for logistic regression we use a different cost function known as the cross entropy error function :

$$L(W) = \begin{cases} -\log h(x^i) & \text{if } y = 1 \\ -\log(1 - h(x^i)) & \text{if } y = 0 \end{cases}$$



Notice if $y = 0$ and our hypothesis predicts a value:

- Close to 0 then our cost is very low ($-\log(1 - 0.01) = -\log(0.99)$).
- However, if $y = 0$ and our hypothesis predicts a value close to 1 then our cost will be very high ($-\log(1 - 0.99) = -\log(0.01)$).

Cross Entropy Loss Function

- ▶ We can more concisely represent the error or loss function as for a single training example X^i as...

$$L(W) = -y \log(h(x^i)) - (1 - y) \log(1 - h(x^i))$$

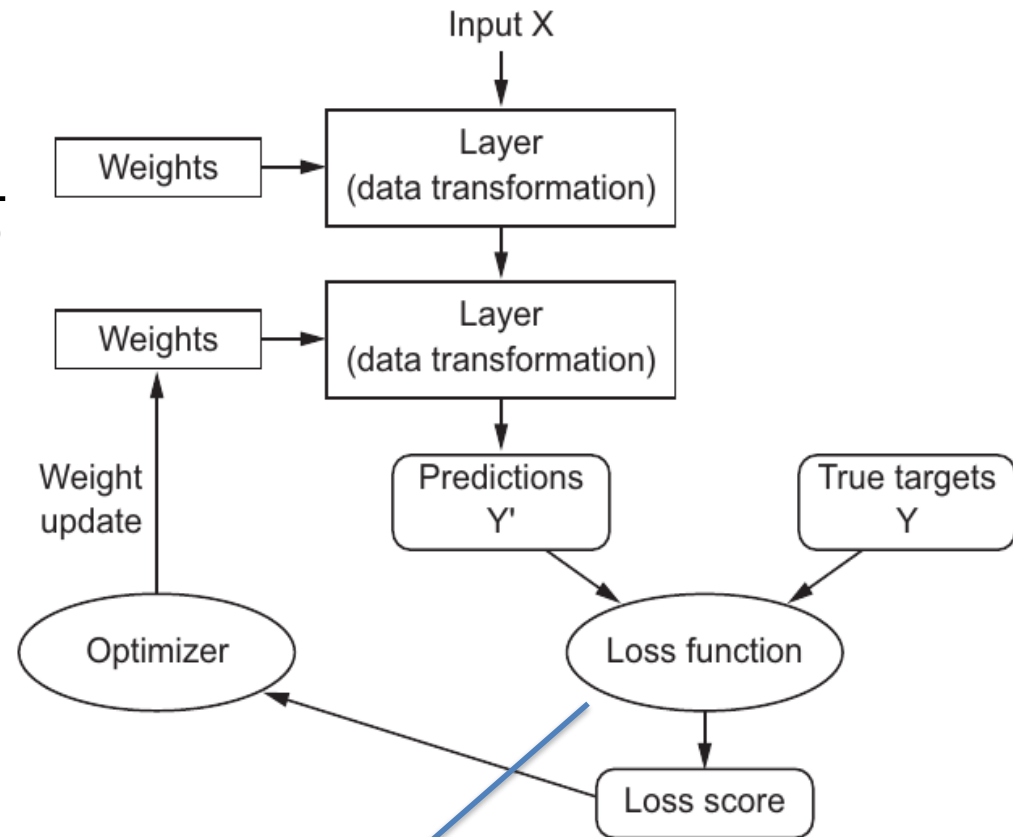
- ▶ The loss function above defines how well our hypothesis is doing on a single training example.
- ▶ However, we need a cost function, which will quantify how well our hypothesis (with current weights and bias) is doing on the entire training set.

$$C(W, b) = \frac{1}{m} \sum_{i=1}^m L(W)$$

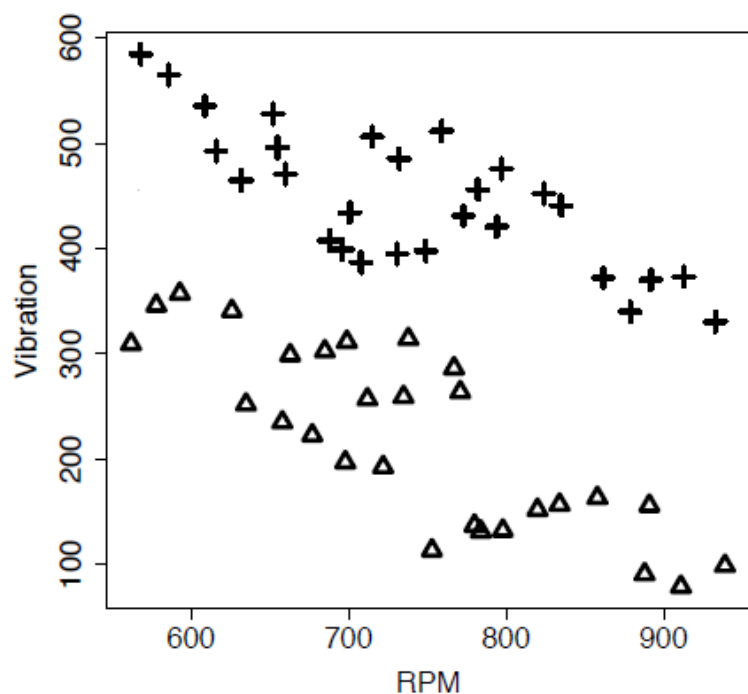
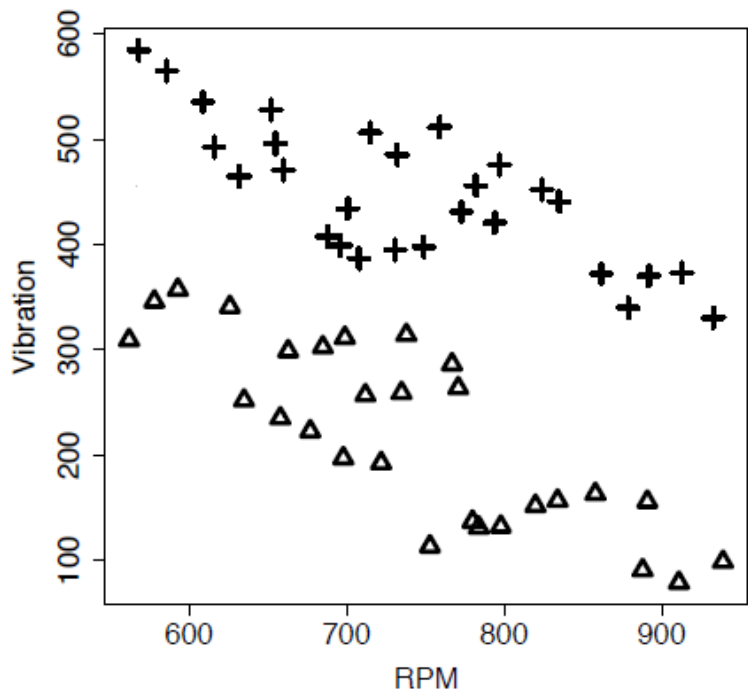
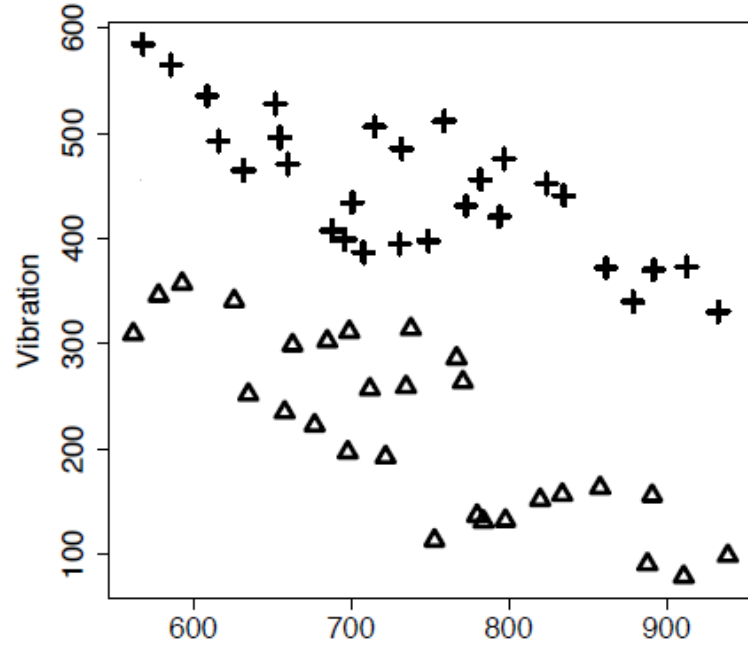
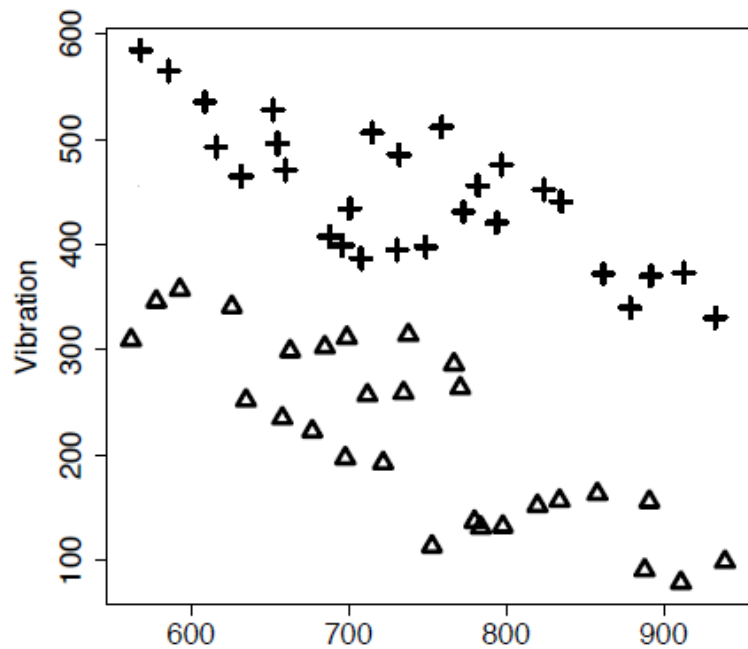
At this stage we have a cost function for logistic regression. Let's now apply gradient descent in the same way we did with linear and multi-variate linear regression.

Now we have our model and our new loss function

$$\text{logistic}(X\lambda^T + b) = \frac{1}{1 + e^{-X\lambda^T + b}}$$



$$\mathcal{L}(W) = -y \log(h(x^i)) - (1 - y) \log(1 - h(x^i))$$



Updating Weight in Logistic Regression

- As we did with linear regression we can now use gradient descent to find the optimal values for each λ_i and the bias b
- Repeat {

$$\lambda_j = \lambda_j - \alpha \frac{\partial}{\partial \lambda_j} C(W, b)$$
$$b = b - \alpha \frac{\partial}{\partial b} C(W, b)$$

}

Where $\frac{\partial}{\partial \lambda_j} C(W, b) = \frac{1}{2m} \sum_{i=1}^m ((h(x^i) - y^i))(x_j^i)$

$$\frac{\partial}{\partial b} C(W, b) = \frac{1}{2m} \sum_{i=1}^m (h(x^i) - y^i)$$

Updating Weight in Logistic Regression

$$\frac{\partial}{\partial \lambda_j} C(W, b) = \frac{1}{2m} \sum_{i=1}^m ((h(x^i) - y^i))(x_j^i)$$

To express this in simpler terms, let's assume a concrete example. Let's say we want to get the partial derivative for the cost function with respect to the weight/coefficient λ_1 . Therefore, in this case $j = 1$ (think of this as feature 1 from our training set).

For each training example (of which there are m), we determine:

- The difference between the predicted value for the i^{th} training example ($h(x^i)$) and the actual value (y^i).
- We multiply the result by the value of the first feature (because $j = 1$) for the i^{th} training example.


Finally we add up the result for each training example and average by dividing by $2 \cdot m$.

- In the example we illustrate on the next few slides we are going to show how to apply the gradient descent update rule for λ_1

$$\lambda_1 = \lambda_1 - \alpha \frac{\partial}{\partial \lambda_1} C(W, b)$$

- Therefore, in our example below let's assume that the current value of λ_1 is 0.25 and the value of α is 0.1. Therefore, our update for λ_1 would look like:

$$\lambda_1 = 0.25 - 0.1 \frac{\partial}{\partial \lambda_1} C(W, b)$$



$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

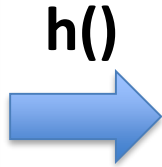
Over the next few slides we will illustrate how to calculate $\frac{\partial}{\partial \lambda_1} C(W, b)$ for the simple dataset on the right, which has 4 rows and 2 features.

52	173
59	184
67	194
73	211

- In this example let's assume that we want to update λ_1
- Therefore, below we calculate the partial derivative for λ_1

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((\underline{h(x^i)} - y^i))(x_1^i)$$

52	173
59	184
67	194
73	211



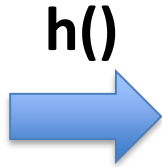
$p_1 = \text{logistic}(52 \lambda_1 + 173 \lambda_2 + b)$
 $p_2 = \text{logistic}(59 \lambda_1 + 184 \lambda_2 + b)$
 $p_3 = \text{logistic}(67 \lambda_1 + 194 \lambda_2 + b)$
 $p_4 = \text{logistic}(73 \lambda_1 + 211 \lambda_2 + b)$

Push each feature vector through our model $h(x) = \frac{1}{1 + e^{-x\lambda^T + b}}$. Collect each prediction p_1 to p_4

- In this example let's assume that we want to update λ_1
- Therefore, below we calculate the partial derivative for λ_1

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(\underline{x_1^i})$$

52	173
59	184
67	194
73	211



$p_1 = \text{logistic}(52 \lambda_1 + 173 \lambda_2 + b)$
 $p_2 = \text{logistic}(59 \lambda_1 + 184 \lambda_2 + b)$
 $p_3 = \text{logistic}(67 \lambda_1 + 194 \lambda_2 + b)$
 $p_4 = \text{logistic}(73 \lambda_1 + 211 \lambda_2 + b)$



$d_1 = p_1 - y_1$
 $d_2 = p_2 - y_2$
 $d_3 = p_3 - y_3$
 $d_4 = p_4 - y_4$

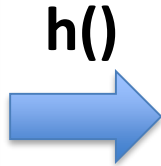
Push each feature vector through our model $h(x) = \frac{1}{1 + e^{-x\lambda^T + b}}$. Collect each prediction p_1 to p_4

Get the difference between each predicted result and the actual true target regression value. In our notation above p_i is the prediction for i th training instance and y_i is the true target regression value for the i th instance.

- In this example lets assume that we want to update λ_1
- Therefore, below we calculate the partial derivitave for λ_1

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(\underline{x_1^i})$$

52 173
59 184
67 194
73 211



$p_1 = \text{logistic}(52 \lambda_1 + 173 \lambda_2 + \mathbf{b})$
 $p_2 = \text{logistic}(59 \lambda_1 + 184 \lambda_2 + \mathbf{b})$
 $P_3 = \text{logistic}(67 \lambda_1 + 194 \lambda_2 + \mathbf{b})$
 $p_4 = \text{logistic}(73 \lambda_1 + 211 \lambda_2 + \mathbf{b})$



$d_1 = p_1 - y_1$
 $d_2 = p_2 - y_2$
 $d_3 = P_3 - y_3$
 $d_4 = p_4 - y_4$



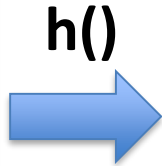
$s_1 = d_1 * 52$
 $s_2 = d_2 * 59$
 $s_3 = d_3 * 67$
 $s_4 = d_4 * 73$

Multiply the difference between the predicted result and actual result for the ith training instance by the ith value for feature 1

- In this example lets assume that we want to update λ_1
- Therefore, below we calculate the partial derivitave for λ_1

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

52 173
59 184
67 194
73 211



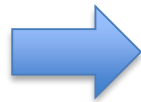
$$\begin{aligned} p_1 &= \text{logistic}(52 \lambda_1 + 173 \lambda_2 + \mathbf{b}) \\ p_2 &= \text{logistic}(59 \lambda_1 + 184 \lambda_2 + \mathbf{b}) \\ p_3 &= \text{logistic}(67 \lambda_1 + 194 \lambda_2 + \mathbf{b}) \\ p_4 &= \text{logistic}(73 \lambda_1 + 211 \lambda_2 + \mathbf{b}) \end{aligned}$$



$$\begin{aligned} d_1 &= p_1 - y_1 \\ d_2 &= p_2 - y_2 \\ d_3 &= p_3 - y_3 \\ d_4 &= p_4 - y_4 \end{aligned}$$



$$\begin{aligned} s_1 &= d_1 * 52 \\ s_2 &= d_2 * 59 \\ s_3 &= d_3 * 67 \\ s_4 &= d_4 * 73 \end{aligned}$$



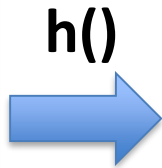
$$\frac{\partial}{\partial \lambda_1} C(W, b) = (s_1 + s_2 + s_3 + s_4) / (2 * 4)$$

Now add up the result for each instance and divide by $2 * m$
(remember m is the number of rows in the training set)

- In this example let's assume that we want to update λ_2
- Therefore, below we calculate the partial derivative for λ_2

$$\frac{\partial}{\partial \lambda_2} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_2^i)$$

52 173
59 184
67 194
73 211



$$\begin{aligned} p_1 &= \text{logistic}(52 \lambda_1 + 173 \lambda_2 + b) \\ p_2 &= \text{logistic}(59 \lambda_1 + 184 \lambda_2 + b) \\ p_3 &= \text{logistic}(67 \lambda_1 + 194 \lambda_2 + b) \\ p_4 &= \text{logistic}(73 \lambda_1 + 211 \lambda_2 + b) \end{aligned}$$



$$\begin{aligned} d_1 &= p_1 - y_1 \\ d_2 &= p_2 - y_2 \\ d_3 &= p_3 - y_3 \\ d_4 &= p_4 - y_4 \end{aligned}$$



$$\begin{aligned} s_1 &= d_1 * 173 \\ s_2 &= d_2 * 184 \\ s_3 &= d_3 * 194 \\ s_4 &= d_4 * 211 \end{aligned}$$



$$\frac{\partial}{\partial \lambda_2} C(W, b) = (s_1 + s_2 + s_3 + s_4) / (2 * 4)$$

Notice here we try to calculate the partial derivative for λ_2 . The only thing that changes is that we multiple the difference between the predicted and actual by feature 2 values.

Pseudocode

$$\frac{\partial}{\partial \lambda_1} \mathcal{C}(W, b) = \frac{1}{2m} \sum_{i=0}^m ((\underline{h(x^i)} - y^i))(x_1^i)$$

Lets again assume we have a simple dataset with two features.

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for i = 1 to m:

$$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$$

The variable h^i is output of our hypothesis function when we input the i th training instance.

Pseudocode

$$\frac{\partial}{\partial \lambda_1} \mathcal{C}(W, b) = \frac{1}{2m} \sum_{i=0}^m ((\underline{h(x^i)} - y^i))(x_1^i)$$

Lets again assume we have a simple dataset with two features.

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for i = 1 to m:

$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$

cost += - [$(y^i)(\log(h^i)) + (1-y^i)(\log(1-h^i))$]

This is our negative log based loss function (cross entropy loss function). The purpose of the cost variable is to keep track of the average cost across all training examples for the current values of λ and b

Pseudocode

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((\underline{h(x^i)} - y^i))(x_1^i)$$

Lets again assume we have a simple dataset with **two features**.

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for i = 1 to m:

$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$

cost += - [$(y^i)(\log(h^i)) + (1-y^i)(\log(1-h^i))$]

err = $h^i - y^i$

Calculate the difference between the ith instance true value and the predicted value.

Pseudocode

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

Lets again assume we have a simple dataset with **two features**.

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for i = 1 to m:

$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$

cost += - [$(y^i)(\log(h^i)) + (1-y^i)(\log(1-h^i))$]

err = $h^i - y^i$

$d\lambda_1 += x_1^i(\text{err})$

$d\lambda_2 += x_2^i(\text{err})$

$db += \text{err}$

Here we add up the partial derivative values for λ_1 and λ_2 and b . Remember, these are added up for every training example and will then be averaged across all training examples.

Pseudocode

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

Lets again assume we have a simple dataset with two features.

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for i = 1 to m:

$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$

cost += - [$(y^i)(\log(h^i)) + (1-y^i)(\log(1-h^i))$]

err = $h^i - y^i$

$d\lambda_1 += x_1^i(\text{err})$

$d\lambda_2 += x_2^i(\text{err})$

$db += \text{err}$

It is worth noting that for the bias we simply have to add the error encountered for error training example and average.

$$\frac{\partial}{\partial b} C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m (h(x^i) - y^i)$$

Pseudocode

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

Lets again assume we have a simple dataset with two features.

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for i = 1 to m:

$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$

cost += - [$(y^i)(\log(h^i)) + (1-y^i)(\log(1-h^i))$]

err = $h^i - y^i$

$d\lambda_1 += x_1^i(\text{err})$

$d\lambda_2 += x_2^i(\text{err})$

$db += \text{err}$

Next we average the partial derivative values

cost = cost/m , $d\lambda_1 = d\lambda_1/(m*2)$, $d\lambda_2 = d\lambda_2/(m*2)$, $db = db/(m*2)$

Lets again assume we have a simple dataset with two features.

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for i = 1 to m:

$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$

cost += - [$(y^i)(\log(h^i)) + (1-y^i)(\log(1-h^i))$]

err = $h^i - y^i$

$d\lambda_1 += x_1^i(\text{err})$

$d\lambda_2 += x_2^i(\text{err})$

$db += \text{err}$

Finally we update the current values for λ and b according to our gradient descent rule.

cost = cost/m , $d\lambda_1 = d\lambda_1/(m*2)$, $d\lambda_2 = d\lambda_2/(m*2)$, $db = db/(m*2)$

$\lambda_1 = \lambda_1 - \alpha d\lambda_1$

$\lambda_2 = \lambda_2 - \alpha d\lambda_2$

$b = b - \alpha db$

Lets again assume we have a simple dataset with two features.

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for $i = 1$ to m :

$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$

$\text{cost} += - [(y^i)(\log(h^i)) + (1-y^i)(\log(1-h^i))]$

$\text{err} = h^i - y^i$

$d\lambda_1 += x_1^i(\text{err})$

$d\lambda_2 += x_2^i(\text{err})$

$db += \text{err}$

You should be aware that this code demonstrates just one collective update of the parameters. That is, just one single iteration of gradient descent. However, this will need to be repeated many times (as we did with linear regression)

$\text{cost} = \text{cost}/m$, $d\lambda_1 = d\lambda_1/(m*2)$, $d\lambda_2 = d\lambda_2/(m*2)$, $db = db/(m*2)$

$\lambda_1 = \lambda_1 - \alpha d\lambda_1$

$\lambda_2 = \lambda_2 - \alpha d\lambda_2$

$b = b - \alpha db$

Lets again assume we have a simple dataset with two features

cost = 0, $d\lambda_1 = 0$, $d\lambda_2 = 0$, $db = 0$

for i = 1 to m:

$h^i = \text{logistic}(\lambda_1 x_1^i + \lambda_2 x_2^i + b)$

cost += - [$(y^i)(\log(h^i)) + (1-y^i)(\log(1-h^i))$]

err = $h^i - y^i$

$d\lambda_1 += x_1^i(\text{err})$

$d\lambda_2 += x_2^i(\text{err})$

$db += \text{err}$

cost = cost/m , $d\lambda_1 = d\lambda_1/(m*2)$

$\lambda_1 = \lambda_1 - \alpha d\lambda_1$

$\lambda_2 = \lambda_2 - \alpha d\lambda_2$

$b = b - \alpha db$

There are some issues with this code. It depends heavily on the use of for loops.

1. We have a for loop that iterates across m training examples.
2. If we have a large number of features we would also need a for loop for calculating each of the partial derivatives.
3. We will need an outer for loop to iterate through each iteration of Gradient Descent.

A 3 tier nested for loop is highly inefficient.

Contents

1. Introduction to Linear Regression
2. Applying Gradient Decent to Linear Regression
3. Multi-Variate Linear Regression
4. Review of Matrices and Broadcasting in Python
5. Logistic Regression
6. Vectorized Logistic Regression
7. Neural Networks

Vectorised Version of Logistic Regression

Let's assume we have a training set with m rows and n columns (features).

$$X = \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix}$$

We create a matrix \mathbf{X} as follows, where each **column is a row from our training** dataset (we are just getting the transpose of the original matrix). For example, the first column contains all n features from the first row of our dataset.

$$\begin{bmatrix} x_1^1 & x_2^1 & \cdots & x_n^1 \\ \vdots & \ddots & & \vdots \\ x_1^m & x_2^m & \cdots & x_n^m \end{bmatrix}$$



$$\begin{bmatrix} x_1^1 & \cdots & x_1^m \\ x_2^1 & \cdots & x_2^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix}$$

Vectorised Version of Logistic Regression

Let's assume we have a training set with m rows and n columns (features).

$$X = \begin{bmatrix} x_1^1 & \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \cdots & x_n^m \end{bmatrix}$$

We create a matrix **X** as follows, where each **column is a row from our training** dataset (we are just getting the transpose of the original matrix). For example, the first column contains all n features from the first row of our dataset.

$$W = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix}$$

Let's also assume that we have a **column vector W** that contains all lambda values as shown. Clearly in our problem there are n features in the dataset.

Finally, let's assume we have a **row vector Y** that contains the actual regression value for each training instance i .

$$Y = [y^1, y^2, \dots y^m]$$

Vectorised Version of Logistic Regression

- The first step using vectors is to perform vector matrix multiplication as follows (notice this allows us to multiply the vector of lambda weights by each training example and add the bias):

$$A = W^T X + b$$

- The resulting vector A will look like this:

$$A = \langle a^1, a^2, \dots, a^m \rangle$$

- Where a^1 is $x_1^1 \lambda_1 + x_2^1 \lambda_2 + \dots x_n^1 \lambda_m + b$
- To obtain the predicted output for the current values of λ and b we can do the following.

$$H = \text{logistic}(A)$$

$$A = W^T X + b$$

$$[\lambda_1, \lambda_2, \dots \lambda_n] \begin{bmatrix} x_1^1 & x_1^2 \cdots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & x_n^2 \cdots & x_n^m \end{bmatrix} + b$$

$$\frac{\partial}{\partial b} C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m (\underline{h(x^i)} - y^i)$$

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((\underline{h(x^i)} - y^i))(x_1^i)$$

$$A = W^T X + b$$

$$H = \text{logistic}(A)$$

In this example we push all training instances through out logistic regression model. H is a row vector that will contain the output for all m instances. Remember every value in H will be between 0 and 1.

$$\frac{\partial}{\partial b} C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m (\underline{h(x^i)} - y^i)$$

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((\underline{h(x^i)} - y^i))(x_1^i)$$

$$A = W^T X + b$$

$$H = \text{logistic}(A)$$

$$E = H - Y$$

Notice that E now contains the error we obtained when predicting each of the m training examples.

That is, the difference between the predicted and actual value of Y for each training example.

$$E = [e^1, e^2, \dots, e^m]$$

$$\frac{\partial}{\partial b} C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m (h(x^i) - y^i)$$

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

$$A = W^T X + b$$

$$H = \text{logistic}(A)$$

$$E = H - Y$$

$$db = \frac{1}{2m} \sum E$$

To get the derivate for the bias value then we just add up all the values in E and get the average by dividing by 2*m (the number of training examples). The same as the pseudocode we looked at earlier (remember $db = db/m$).

$$\frac{\partial}{\partial b} C(\lambda, b) = \frac{1}{2m} \sum_{i=1}^m (h(x^i) - y^i)$$

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

$$A = W^T X + b$$

$$H = \text{logistic}(A)$$

$$E = H - Y$$

$$db = \frac{1}{2m} \sum E$$

$$d\lambda = \frac{1}{2m} X E^T$$

This step is more complex. Here, we multiply the matrix X (where each row corresponds to a feature) by the transpose of E (which contains the error for each training example).

What is different about this line is that we are not just calculating the partial derivative for λ_1 but for all λ values.

Notice it is the error E that is the common to the update for each derivative value.

$$\frac{\partial}{\partial \lambda_1} C(W, b) = \frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))(x_1^i)$$

We can visualize this operation as you see below. We multiply the error obtained in each training example, by the value of feature i for each training example. For example, to get the partial derivative for λ^4 we go through each training example and we multiply the error, let's say e^1 (the error obtained for training instance 1) by the value x_4^1 (which is the value of feature 4 for the first training example). We add up the values obtained for each training example and then average to get $d\lambda^4$. This is exactly what we did in the previous code.

$$d\lambda = [d\lambda^1, d\lambda^2, \dots, d\lambda^n] = \frac{1}{2m} \begin{bmatrix} x_1^1 & \dots & x_1^m \\ \vdots & \ddots & \vdots \\ x_n^1 & \dots & x_n^m \end{bmatrix} \begin{bmatrix} e^1 \\ e^2 \\ \vdots \\ e^m \end{bmatrix}$$

$$A = W^T X + b$$

$$H = \text{logistic}(A)$$

$$E = H - Y$$

$$db = \frac{1}{2m} \sum E$$

$$d\lambda = \frac{1}{2m} X E^T$$

$$W = W - (\text{alpha}) (d\lambda)$$

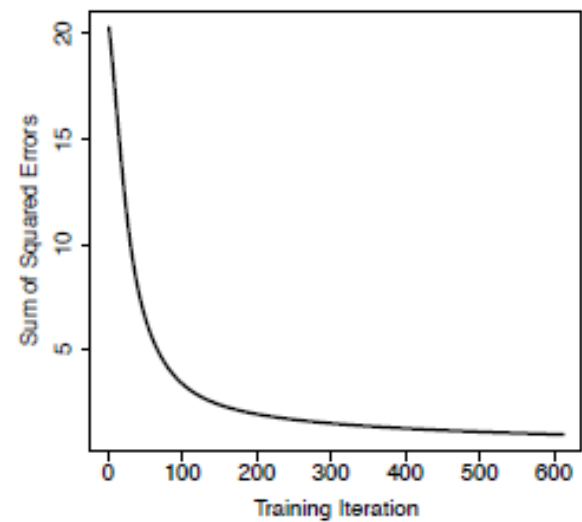
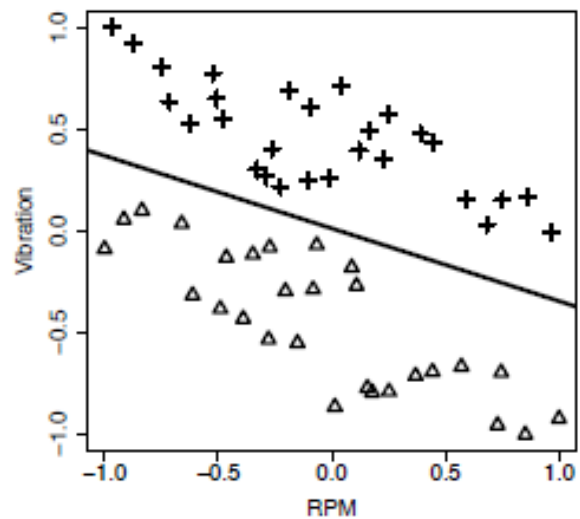
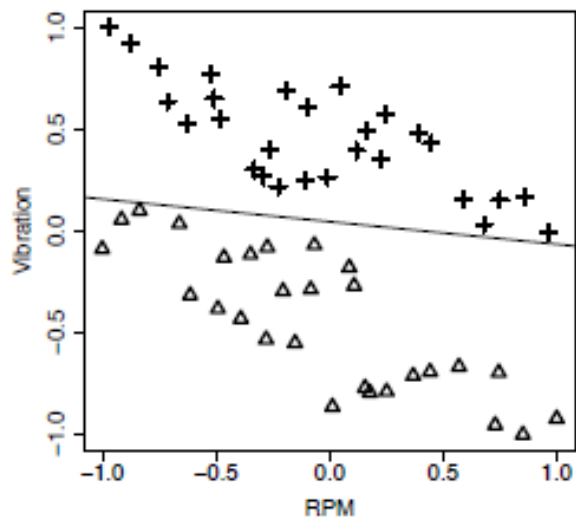
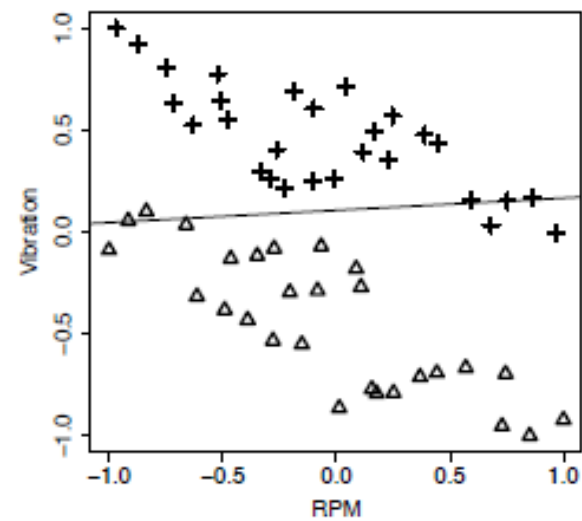
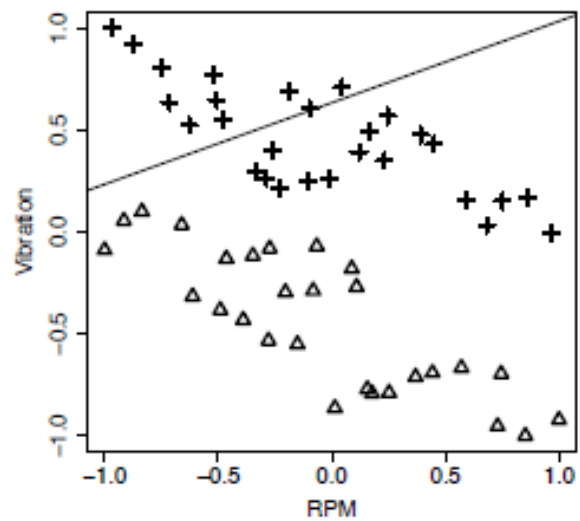
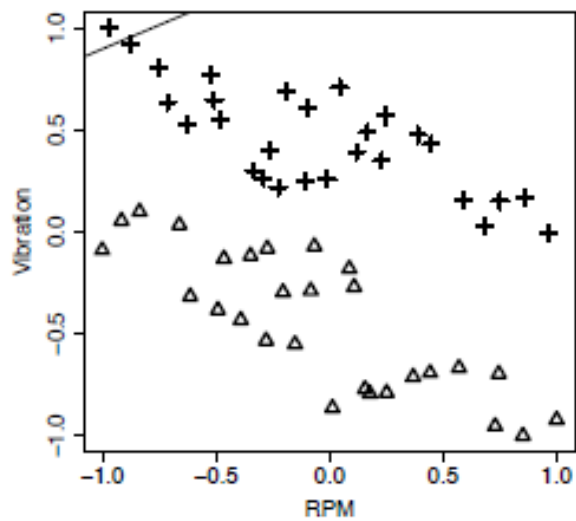
$$b = b - (\text{alpha})(db)$$

Now we have all the derivatives we need and we can directly update the weights and bias according to our gradient descent rule (as normal).

The vectorised code you see here is the same as the code you previously saw.

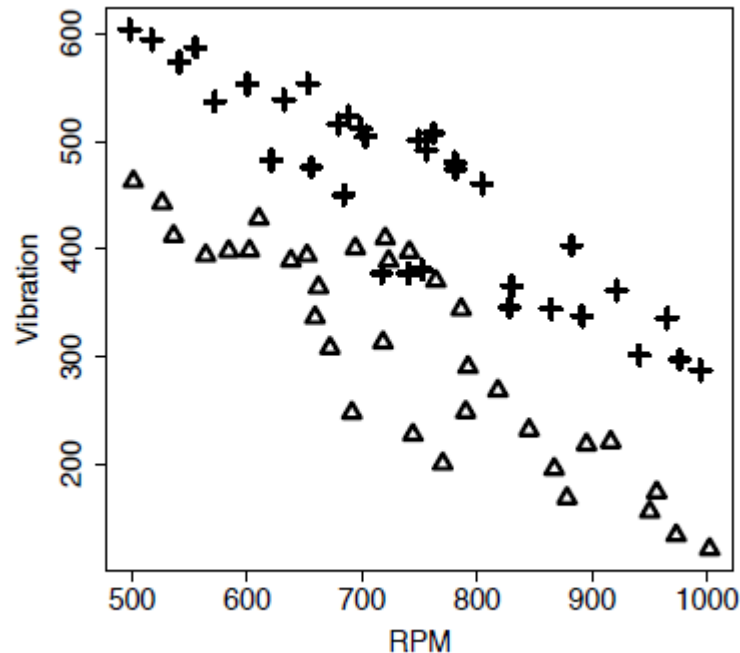
However, it is much much faster.

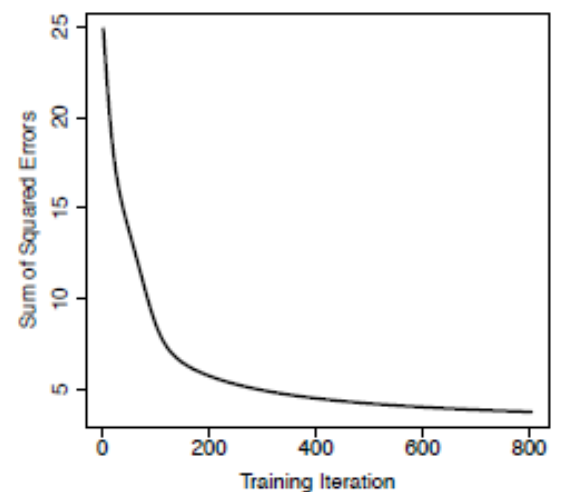
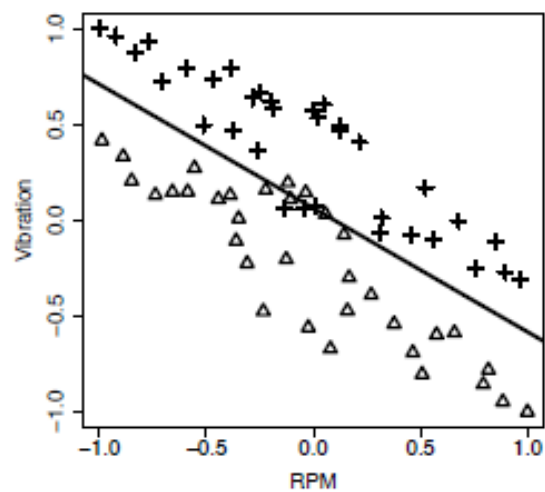
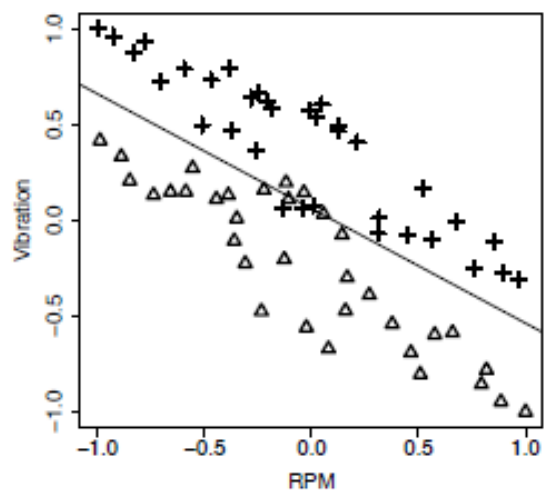
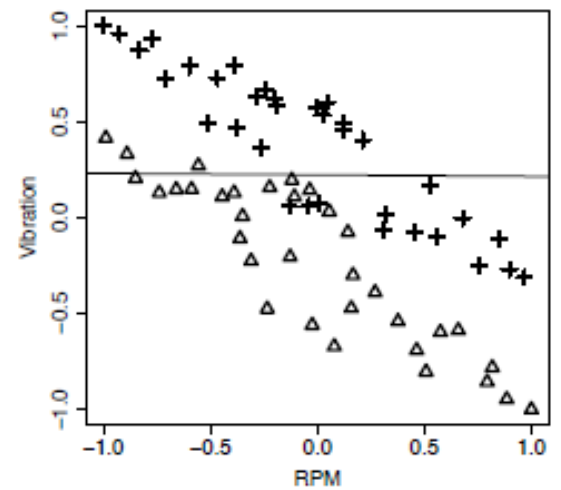
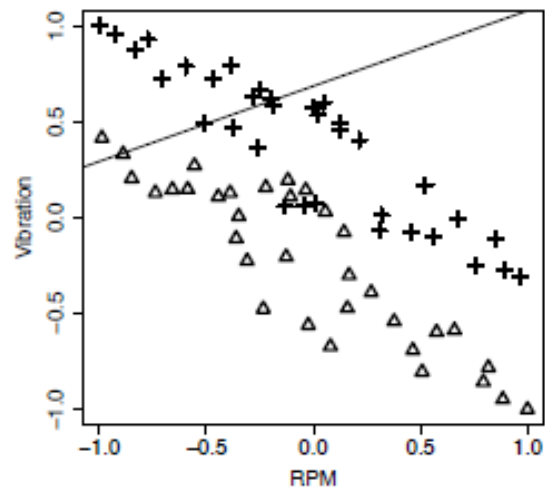
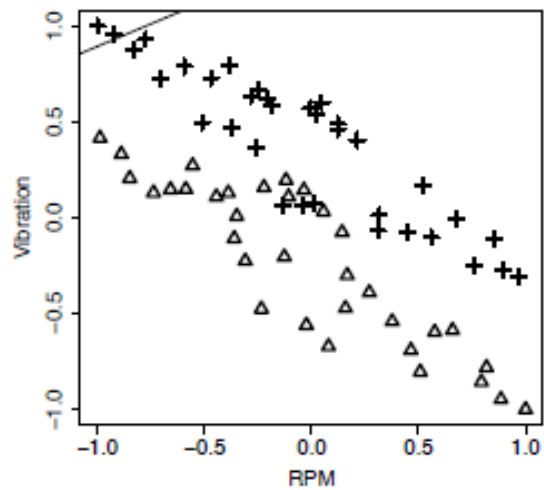
It is worth noting that this is still just one iteration of gradient descent and this code will need to be enclosed in a for loop (it is not possible to completely remove our dependence on for loops.)



Classification for Non-Linearly Separable Datasets

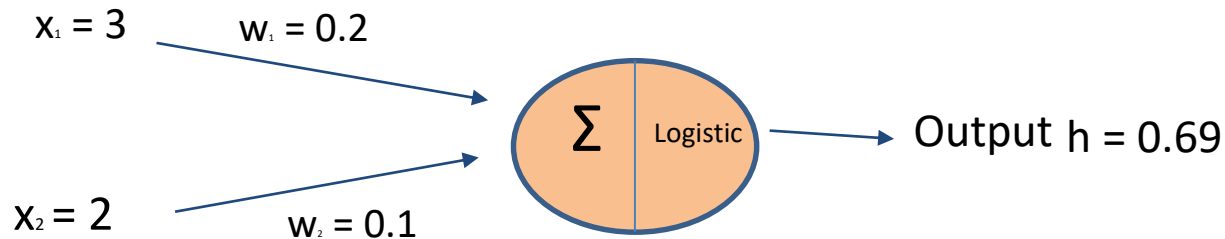
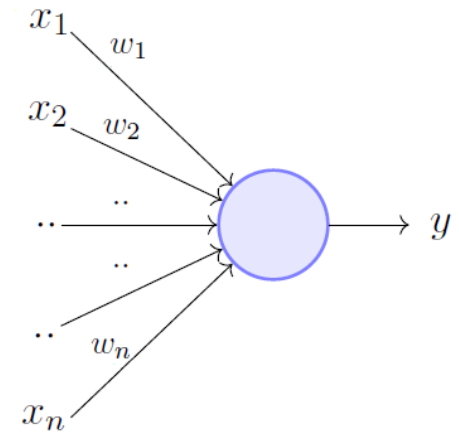
- ▶ The parameters for a logistic regression unit namely the bias and the weights can be initialized to 0.
- ▶ An advantage of logistic regression is that it can still work reasonably well for datasets in which the data is not linearly separable. For example, the scatter plot below shows an altered version of the generator dataset, where there is clear overlap between the two classes (good and faulty).

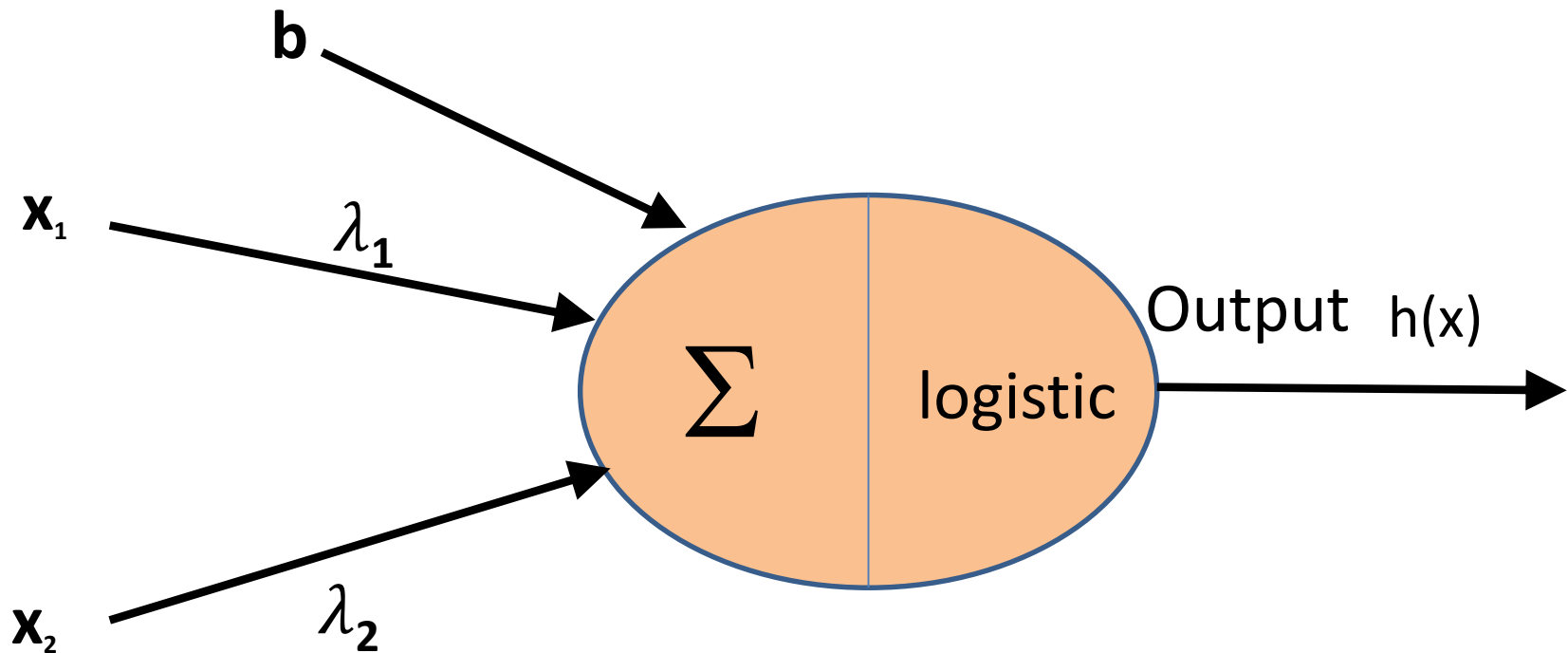




What is a Neuron?

- Artificial neurons are the building blocks of artificial neural networks.
- The neuron receives one or more inputs (which can be feature values) and sum them to produce a prediction.
- More specifically each input is separately weighted. Each weight is multiplied by the input feature value and the resulting sum is passed through a non-linear transformation known as an activation function.

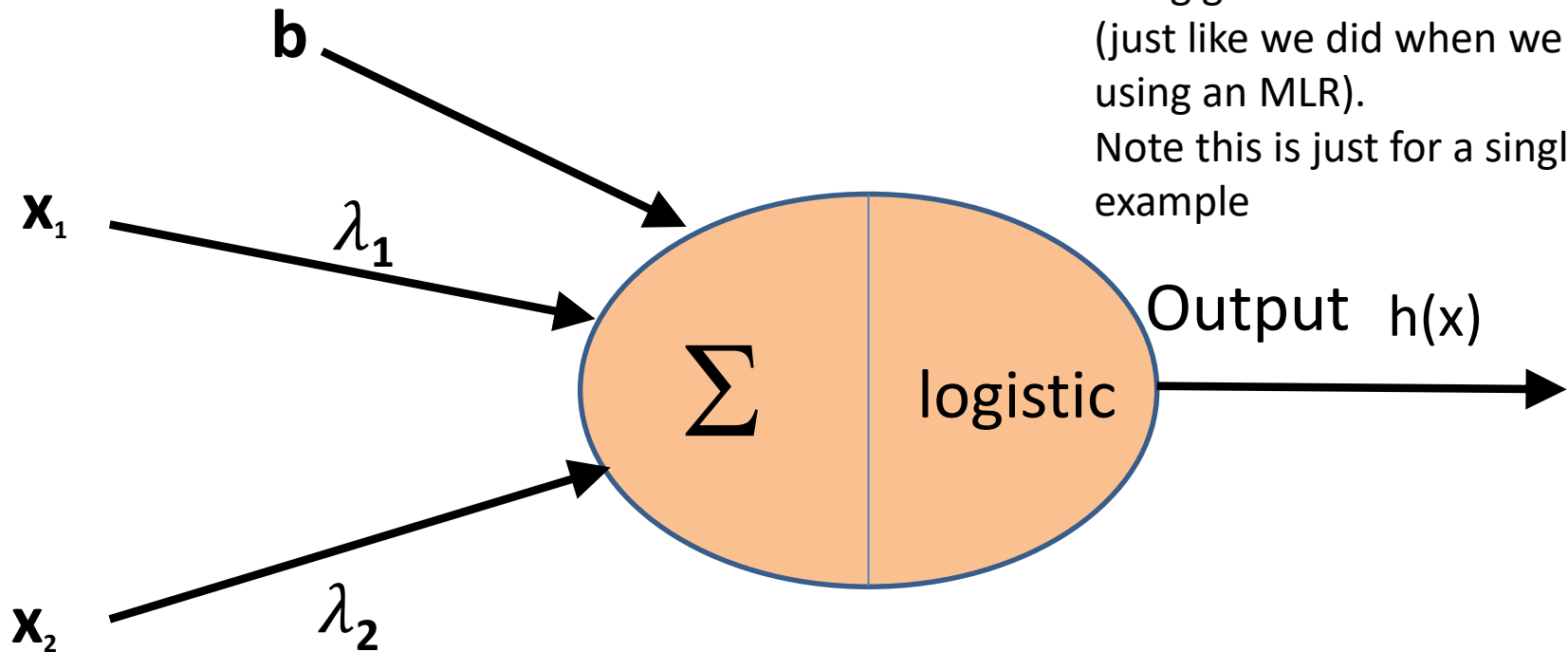




$$L(\lambda) = -y \log(h(x^i)) - (1 - y) \log(1 - h(x^i))$$

For the example x^i this quantifies how good or bad the estimated output from the logistic regression unit was (note this is a value between 0 and 1).

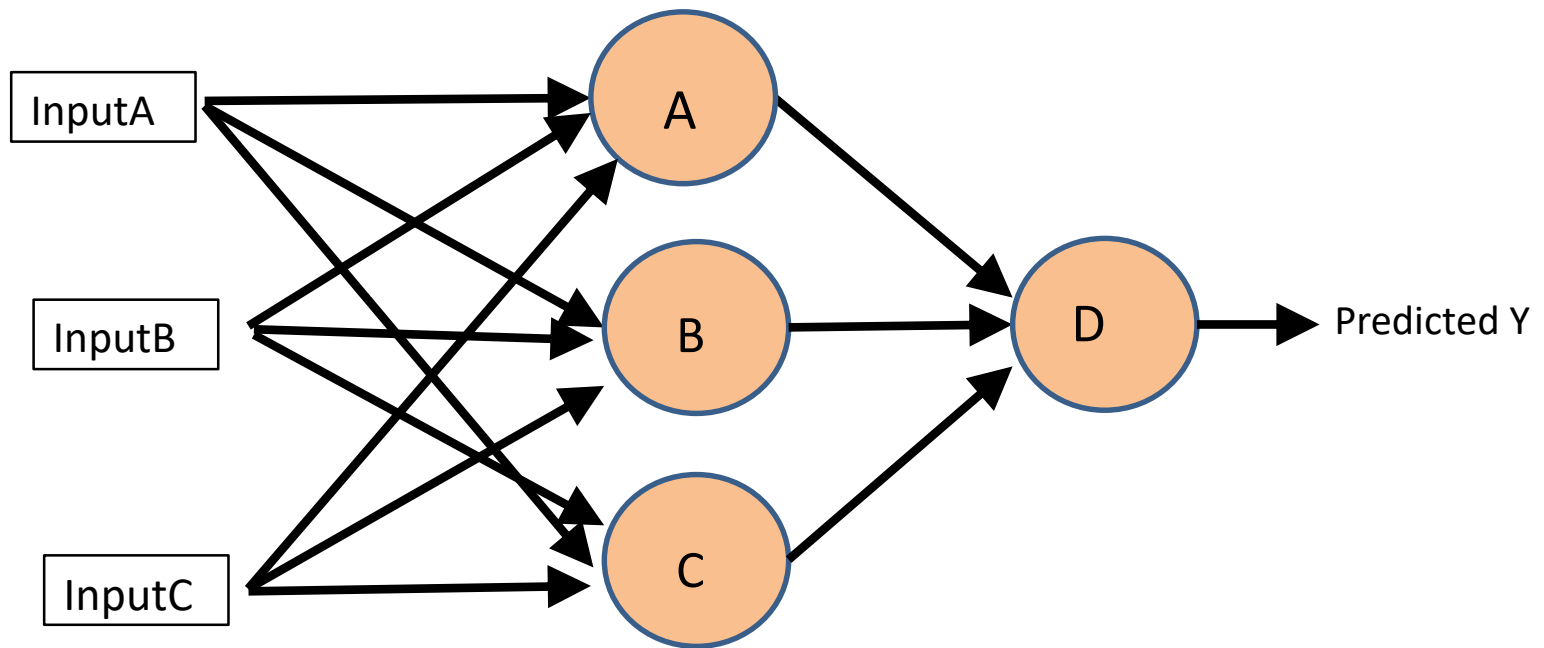
In this step we update the weights lambda and the bias using gradient descent rule (just like we did when we were using an MLR).
Note this is just for a single example



- $\lambda_1 = \lambda_1 - \alpha (x_1) (h(x) - y)$
- $\lambda_2 = \lambda_2 - \alpha (x_2) (h(x) - y)$
- $b = b - \alpha (h(x) - y)$

A Neural Network

- The example, the image below shows a two layer neural network with a single hidden layer (notice the inputs are not included as a layer in neural networks). Notice we are stacking together some of the neurons from the previous slide.

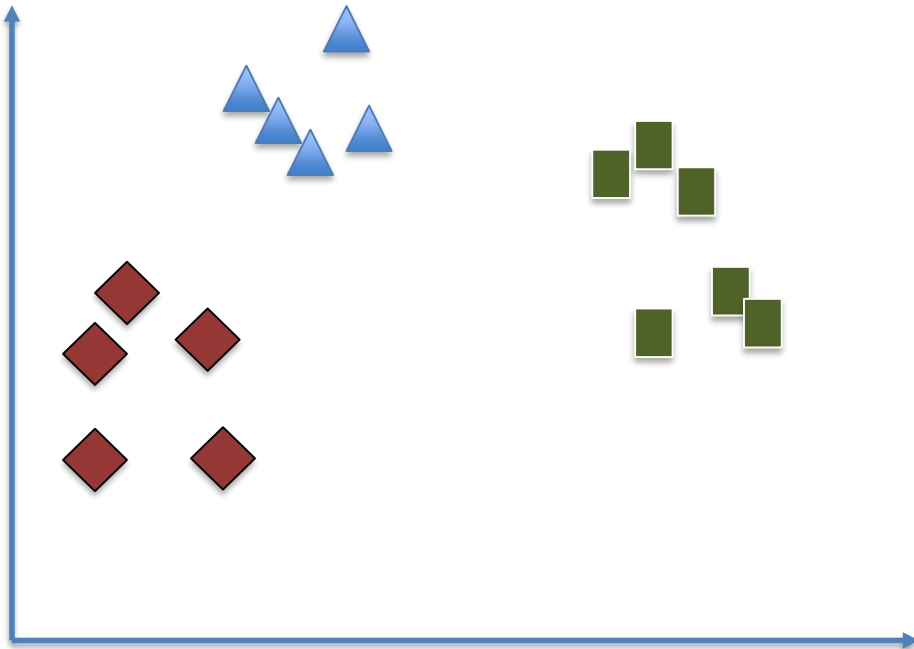


Multiclass Classification

- In the logistic regression problem we have looked at, the problem was one of binary classification. That is, we wanted to determine if a data instance should be classified as one class (0) or another class (1).
- However, as we know multi-class classification involves problems where a data instance can be classed as one of multiple classes (>2 classes).
- There are two main techniques for enabling binary classification algorithms to be extended and used for multi-class classification problems (both of which work by reducing the problem of multiclass classification to multiple binary classification problems)
 - One v's All
 - One v's One

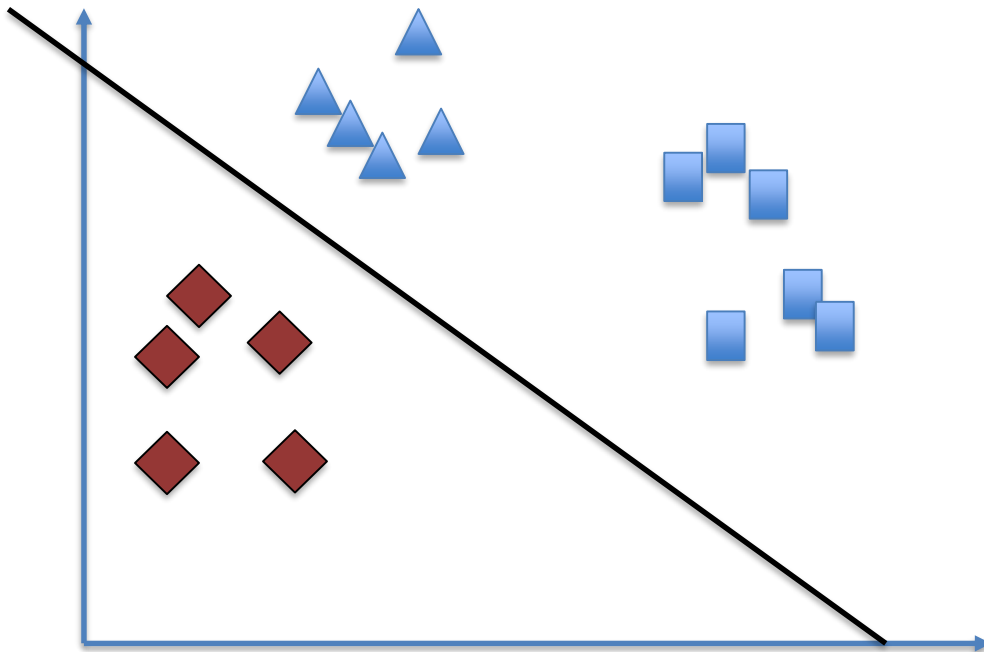
One v's All - Multiclass Classification

- One solution to enable binary classifiers to be used for multi-class classification is a simple method called one v's all (also called one v's rest).
- This approach converts a single multi-class classification problem into **n binary classification problems** where n is the number of classes in the problem.
- For example, in the problem would become three different binary classification problems.



One v's All - Multiclass Classification

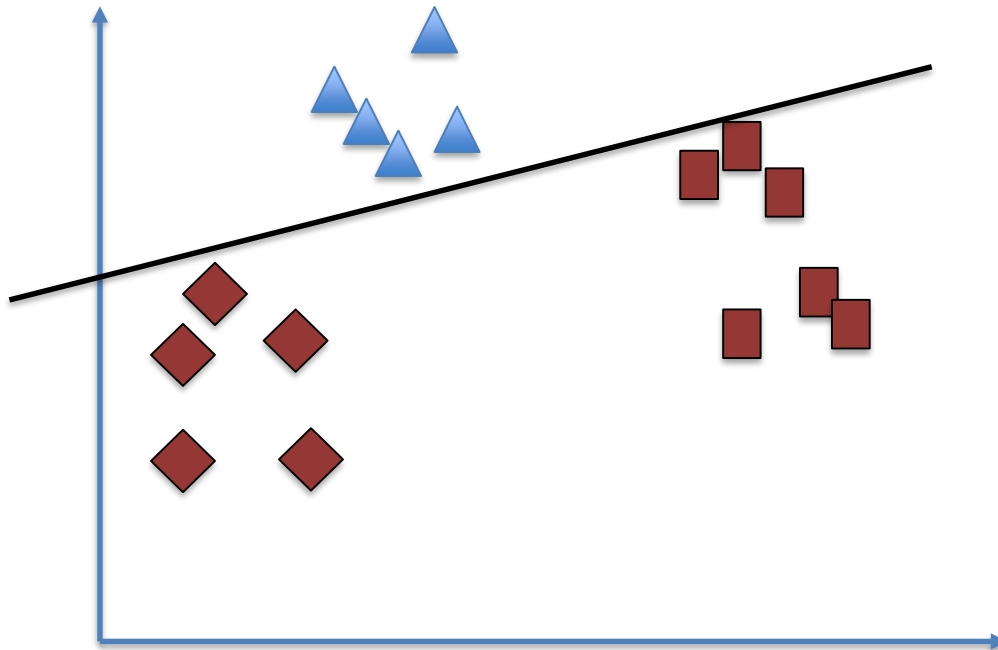
- In one v's all we build three different binary classifiers.
- The objective of the first is to differentiate between Class 1 and all other data instance. All other data instances are labelled as class 2.



- Class 1 – Red
- Class 2 – Blue
- Class 3 - Green

One v's All - Multiclass Classification

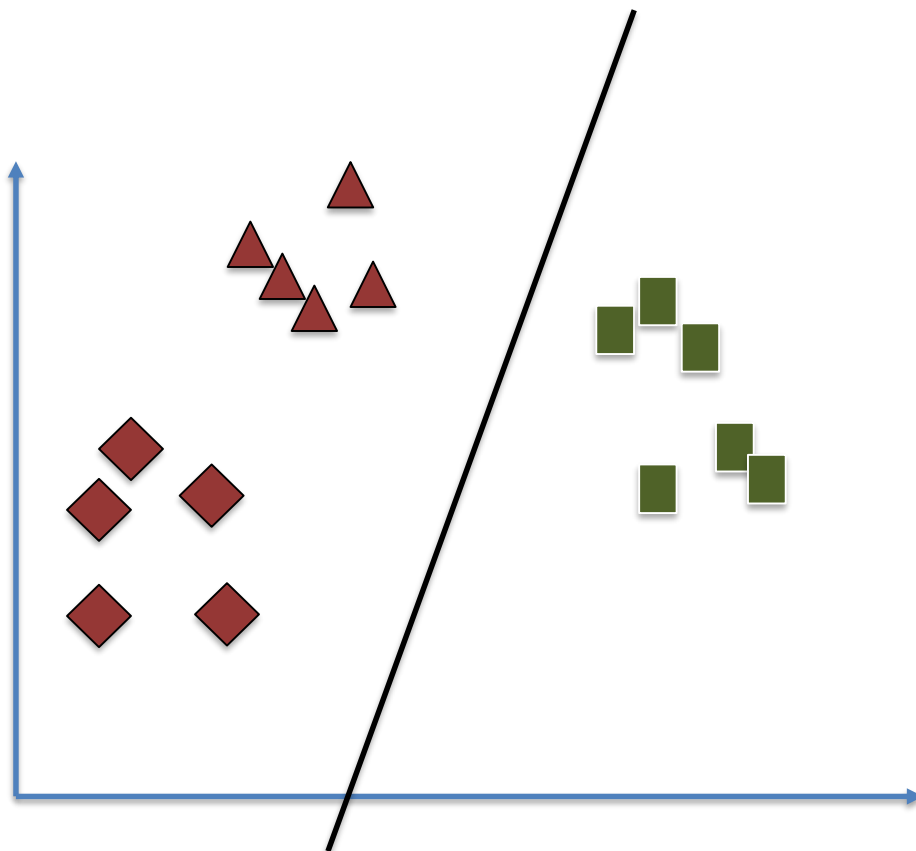
- In one v's all we build three different binary classifiers.
- The objective of the second is to differentiate between Class 2 and all other data instance. All other data instances are labelled as class 1.



- Class 1 – Red
- Class 2 – Blue
- Class 3 - Green

One v's All - Multiclass Classification

- In one v's all we build three different binary classifiers.
- The objective of the third is to differentiate between Class 3 and all other data instance. All other data instances are labelled as class 1.



- Class 1 – Red
- Class 2 – Blue
- Class 3 - Green

Multiclass Classification

- Class 1 – Red
- Class 2 – Blue
- Class 3 - Green

- As we know logistic regression provides us with the probability that a data instance belongs to a specific class.
- Therefore, when we obtain a new data instance (x) to classify we run it through each of the three models and obtain a probability of that instance being classified as class 1, 2, or 3.
 - $P(y = 1 \mid x)$
 - $P(y = 2 \mid x)$
 - $P(y = 3 \mid x)$
- The new data instance x is classed the same as the class with the highest probability value.
- The One v's All approach requires the binary classifier to produce a real-valued confidence (probability) score for its decision.

One v's One Multi-class Classification

- An alternative is called a one v's one multi-class classification approach.
- In this approach you build a binary classifier between every class. Therefore, if you have n classes then you would build $(n * (n-1))/2$ classifiers.
- For example, if we had four classes then we would build $(4 * 3)/2 = 6$ different binary classifiers:
 1. Class1 v Class2
 2. Class1 v Class3
 3. Class1 v Class4
 4. Class2 v Class3
 5. Class2 v Class4
 6. Class3 v Class4
- At prediction time, the class which received the most votes is selected. If we obtain the following predictions from each of our classifiers for a new data instance then we could class it as class 2
 - [2, 1, 4, 2, 2, 3]

- Class 1
- Class 2
- Class 3
- Class 4