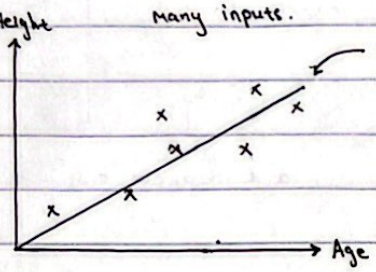
Machine Learning Specialization : Week 1

Supervised vs Unsupervised Machine Learning

Supervised Learning Pt. 1

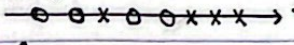
- Supervised learning is when you train your model on input → output pairs until your model can predict the correct output given an input.
- E.g. Handwriting recognition
- Regression: a type of supervised learning problem where you have to predict an output from infinitely many inputs.



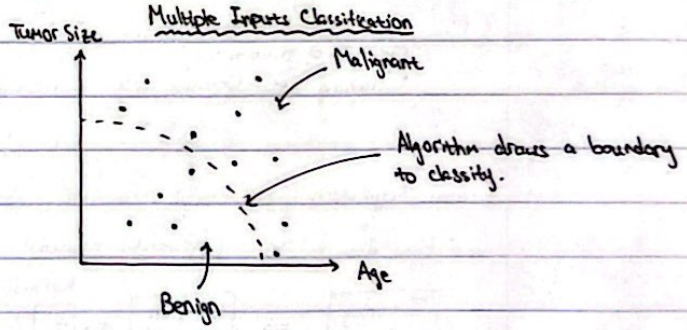
Supervised Learning Pt. 2

- Classification: Another type of supervised learning problem where given an input, your algorithm tries to classify it into a finite number of categories.
- E.g. Breast Cancer detection:

X - Malignant
O - Benign

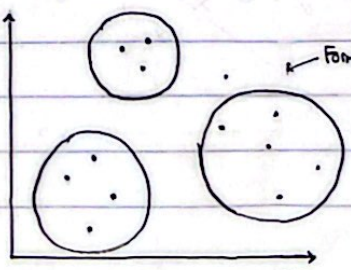


↑ Classifies into {O, X} depending on size



Unsupervised Learning Pt. 1

- On the other hand, unsupervised learning happens when you give your algorithm data and without "correct" output labels and ask it to find patterns on its own.
- Clustering: a type of unsupervised learning which groups data together to form clusters.
- E.g. Grouping similar news articles, categorizing different types of humans by genetic make up, clustering different groups of learners.



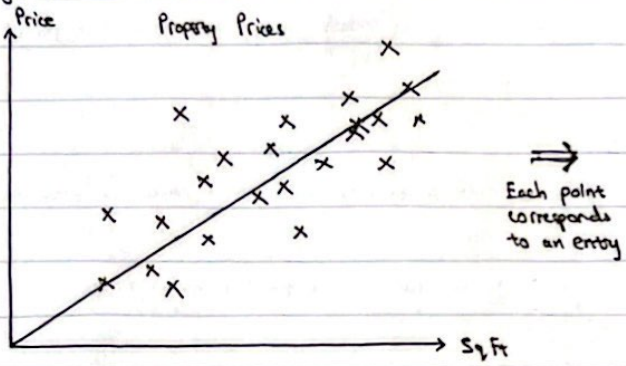
Unsupervised Learning Pt. 2

- Other types of unsupervised clustering algorithms include:
  - Anomaly Detection
  - Dimensionality Reduction - compress data.

Regression Model

Linear Regression Model Pt. 1

- E.g.



⇒ Each point corresponds to an entry

x	y
Sq Ft	Price
200	100,000
...	...

- A linear regression model uses a best-fit line to predict a corresponding output to any infinitely many inputs.
- Notation
  - x: Denotes the input values; input variable
  - y: Denotes output values; output variable
  - (x, y): Denotes an input, output pair.
  - $(x^{(i)}, y^{(i)})$: Denotes the i^{th} input output pair.
  - M - size of training set
  - Training Set: Dataset with correct answers to train model with.

Linear Regression Model Pt. 2

- How is a regression model trained?

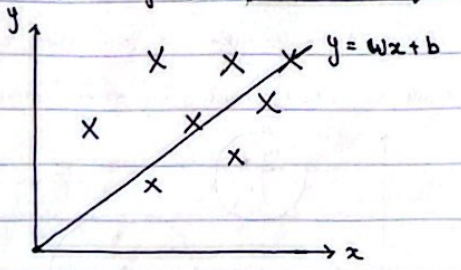
Training Set ⇒ Learning Algorithm ⇒ f (hypothesis)

- Features (x)
- Targets (y)

$x \rightarrow$ f (Prediction for y) $\rightarrow \hat{y}$

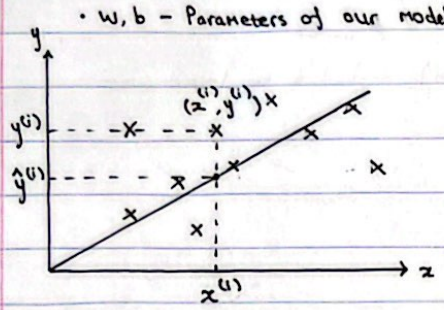
Feature Model

- How do we figure out what f is?
- Linear Regression: $f(x) = wx + b$



Cost Function Formula

- Recall that with our training set, we have our model: $f_{w,b}(x) = wx + b$.
  - w, b – Parameters of our model that can be adjusted to make it more / less accurate.



- There will be an error between target values $y^{(i)}$ and predicted values $\hat{y}^{(i)}$.
- Therefore, to calculate the error of a model, we use the cost function:

$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

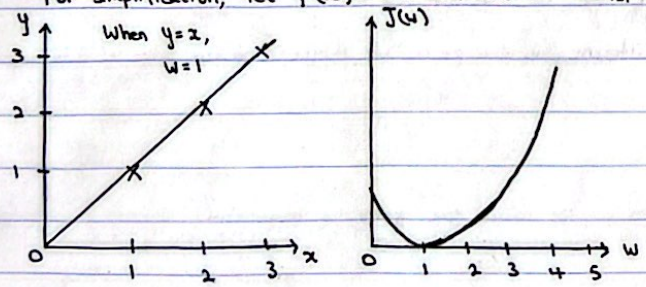
recall m = size of training set

Avg. of errors divided by 2 — Error squared

$$\Rightarrow J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2$$

Cost Function Intuition

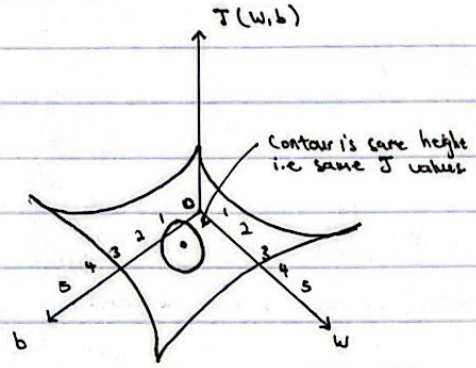
- To find the best fit line $y = f_{w,b}$, we use the cost function $J(w,b)$ and find when it's at a minimum.
- For simplification, let $f(x) = wx$ and $J(w) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_w(x^{(i)} - y^{(i)})^2\right)$, setting parameter b to 0.



- The w that provides the best-fit line is when cost is minimized, i.e. when J is at a minimum.

Visualizing the Cost Function

- Previously, we set $b=0$ to simplify the idea. How do we now get the minimum cost function with b involved?
- We can use a 3D contour model to represent $J(w, b)$, and find the lowest point.



Visualization Examples

- It's difficult to determine the minimum cost manually.
- We do so by using gradient descent.

Train The Model With Gradient Descent

Gradient Descent

- Given a cost function $J(w_1, w_2, \dots, w_n, b)$, we can use gradient descent to find the lowest cost.
- Gradient descent starts at ~~some point~~ arbitrary point on the 3D graph, and "walks" down the steepest path ~~to~~ until it reaches the local minima.
- However, if we start from a different starting point, we might end up with a different local minima.

Implementing Gradient Descent

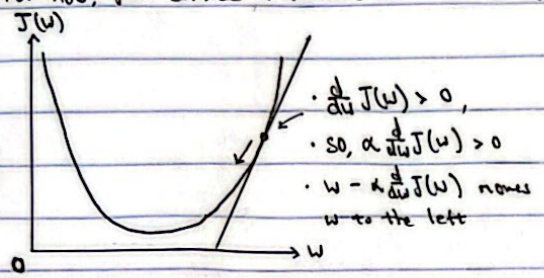
- We slowly adjust parameters w and b by using the following algorithm:

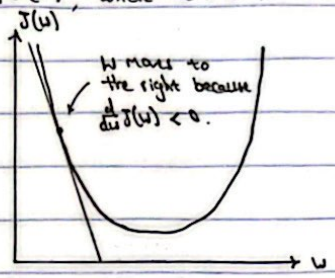
$$w = w - \alpha \frac{d}{dw} J(w,b)$$
$$b = b - \alpha \frac{d}{db} J(w,b)$$

  - α - Learning Rate; how big of a step to take
  - We have to <u>simultaneously</u> update w and b.

Gradient Descent Intuition

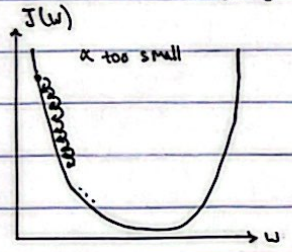
- For now, just consider 1 variable w: $w = w - \alpha \frac{d}{dw} J(w)$, where $\alpha > 0$.

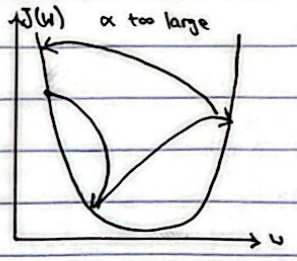




Learning Rate

- Choosing the correct value for α is important:
  - If α is too small, gradient descent will still work but take a very long time
  - If α is too large, gradient descent might overshoot the local minimum.





- Once local min. is found $\alpha \frac{d}{dw} J(w) = 0$ so no more steps taken.
- The size of steps decreases as w approaches the local min. as the slope decreases.

Gradient Descent for Linear Regression

- Repeat until convergence:

$$w = w - \alpha \frac{d}{dw} J(w,b) = w - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right) x^{(i)} \right]$$

$$b = b - \alpha \frac{d}{db} J(w,b) = b - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)} \right) \right]$$

- Above formula is found via differentiation.
- Squared error cost function only has 1 local minimum.

Running Gradient Descent

- Batch Gradient Descent: Algorithm is performed on the whole batch of data.

Machine Learning Specialization : Week 2

Multiple Linear Regression

Multiple Features

- Previously, we looked at using linear regression for problems with only 1 variable (e.g. predicting housing prices with size $[x]$ ft^2). $[y]$
- With multiple variables:

Size ft^2	No. Bedrooms	No. Floors	Age of House	Price ($1000)
2104	3	1	50	40
1416	2	1	20	70
1882	4	2	30	80

Notation

- x_j - The j^{th} feature. E.g. x_1 is size ft^2.
- $\vec{x}^{(i)}$ - The i^{th} row; is a vector row.
- $x_j^{(i)}$ - The i^{th} element for the j^{th} feature.
- n - Number of training features.

Model : $\boxed{f_{w,b}(x) = w_1x_1 + w_2x_2 + \dots + w_nx_n + b}$

- $\vec{w} = [w_1, w_2, \dots, w_n]$
- $\vec{x} = [x_1, x_2, \dots, x_n]$
- b is a constant

$\Rightarrow \boxed{f(x) = \vec{w} \cdot \vec{x} + b}$

Vectorization Pt 1

- To calculate $f_{w,b}(x)$ efficiently, we use vectorization using numpy's built-in dot function:

```
f = np.dot(w, x) + b
```

Vectorization Pt 2

- Behind the scenes, vectorization is efficient because the computer performs calculations in parallel.
- E.g. $\vec{x} \cdot \vec{w}$ to find Model prediction

Gradient Descent : $\vec{w} - \vec{d}$

Without Vectorization:

```
for j in range(len(w)):
    w[j] -= d[j] * alpha
```

vs.

With Vectorization:

```
w -= d * alpha
```

Lab: Python, Numpy and Vectorization

Vectors

- Vectors can be represented using numpy arrays.
- Basic operations: indexing, slicing
- Single Vector Operations: scale by a factor, mean(), sum().
- Vector-vector operations: Adding 2 vectors, dot product

Matrices

- Basically 2D arrays with numpy.
- Index using matrix[row][col].
- array.reshape(3,2): Reshape array to a matrix of 3 rows and 2 colums.
- Slicing: matrix[dimension_1, dim_2, dim_3 ...]

  E.g. array = [[1,2,3],
  [4,5,6],
  [7,8,9]]

  array[1:,1:] # 1st row onwards and 1st col onwards

  $\Rightarrow$ [[5,6],
  [8,9]]

Gradient Descent for Multiple Linear Regression

Notation

- Parameters: $\vec{w} = [w_1, \dots, w_n]$
- Model: $f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$
- Cost Function: $J(\vec{w}, b)$
- Gradient Descent:

$$\begin{cases} w_j = w_j - \alpha \frac{d}{dw_j} J(\vec{w}, b) \\ b = b - \alpha \frac{d}{db} J(\vec{w}, b) \end{cases}$$

Repeat $\Rightarrow$

$$\begin{cases} w_1 = w_1 - \alpha \frac{1}{M} \sum_{i=1}^{M} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_1^{(i)} \\ \dots \\ w_n = w_n - \alpha \frac{1}{M} \sum_{i=1}^{M} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_n^{(i)} \end{cases}$$

$$b = b - \alpha \frac{1}{M} \sum_{i=1}^{M} \left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right)$$

- The normal equation is an alternative to gradient descent for finding $\vec{w}$ and b for linear regression models. It may be used behind the scenes in some libraries.

Gradient ~~Feature~~ Descent in Practice

Feature Scaling Pt 1

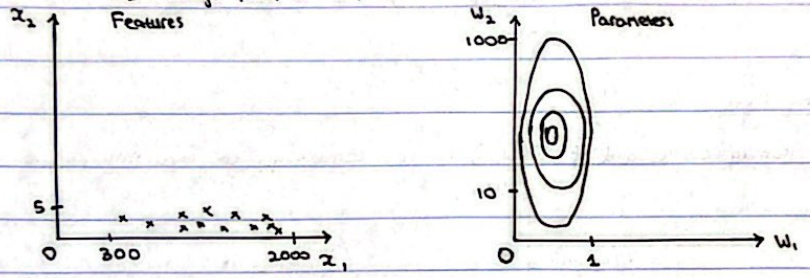
- Gradient descent would work very slowly on a model whose features have very different scales:

E.g. ~~$f_{\vec{w},b}(\vec{x}) = 5w_1 + 5000w_2 + b$~~

price = $w_1x_1 + w_2x_2 + b$

x_1: range from 300 - 2000.

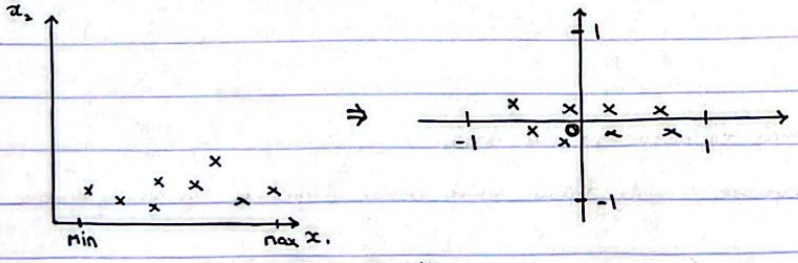
x_2: range from 0 - 5.



- Because of the uneven shape in parameters, gradient descent might take a very long time to find the minimum.
- To get an accurate prediction, we would have to multiply x_1 by a small w_1 since x_1 is large and x_2 by a large w_2.
- We can fix this by scaling the features to the same scale.

Feature Scaling Pt 2

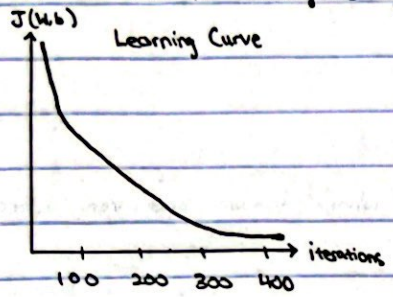
- One way to scale our features is to divide by our max. value.
- Mean Normalization: $\boxed{x_1 = \frac{x_1 - \mu_1}{\text{max} - \text{min}}}$, where μ_1 is the mean.



- Z-Score Normalization: $x_1 = \frac{x_1 - \mu_1}{\sigma_1}$, σ_1 being std. deviation.
- You don't have to scale when your range is somewhere near $-1 \leq x_j \leq 1$.

Checking Gradient Descent for Convergence

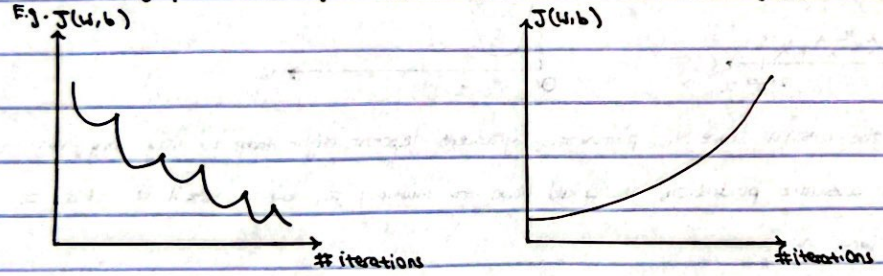
- We can check for convergence by graphing $J(\vec{w}, b)$ by the number of iterations of updates.



- We Automatic Convergence Test: Let ϵ be a very small number, e.g. 10^{-3}. When $J(\vec{w}, b)$ decreases by $\leq \epsilon$ in one iteration we declare convergence.

Choosing the Learning Rate

- We can graph the learning curve, and if $J(\vec{w}, b)$ is not converging, we can try picking smaller α values.



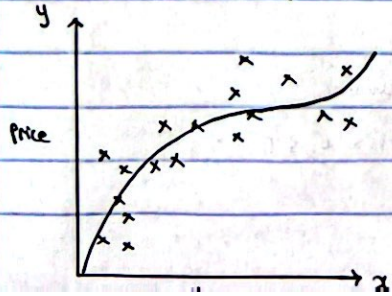
Feature Engineering

- Sometimes, we can increase the accuracy of our models by combining features.
- E.g. If we have x_1 and x_2 as the width and length of a property, we can define $x_3 = x_1 x_2$:

$$f_{\vec{w},b}(\vec{x}) = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

Polynomial Regression

- We can use polynomials to better fit our data.
- When we do it's important to apply feature scaling because polynomials can greatly increase the range



$$f_{\vec{w},b}(\vec{x}) = w_1 x_0 + w_2 x_0^2 + w_3 x^3 + b$$

Lab: Feature Engineering and Polynomial Regression

- We can deal with polynomial features by engineering them. E.g. $x_3 = x_1 x_2$.
- Gradient descent will automatically help us select the better features by assigning them larger parameters.
- In fact, polynomial features that best fit the data will have a linear relationship with it.
- Applying feature scaling increases the effectiveness of gradient descent.

Lab: Linear Regression with Scikit Learn

- Scale / Normalize Data

```
scaler = StandardScaler()
X_norm = scaler.fit_transform(X_train)
```

- Create and Fit Regression Model

```
sgdr = SGDRegressor(max_iter=1000)
sgdr.fit(X_norm, y_train)
```

- Make Predictions

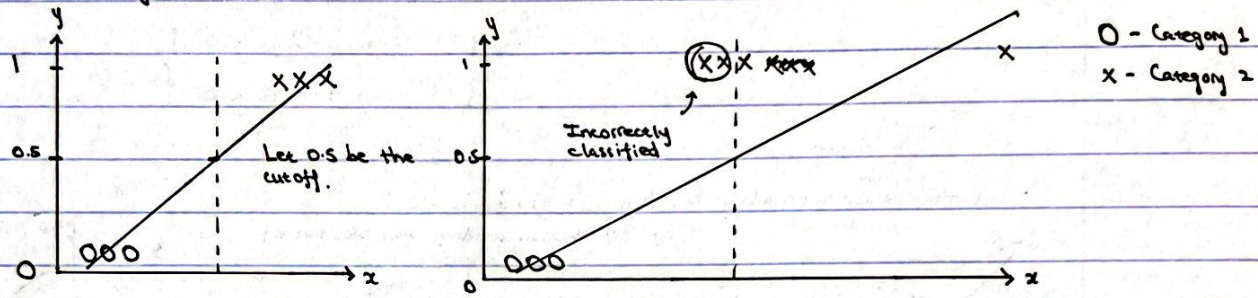
```
y_pred_sgd = sgdr.predict(X_norm)
```

Machine Learning Specialization: Week 3

Classification with Logistic Regression

Motivations

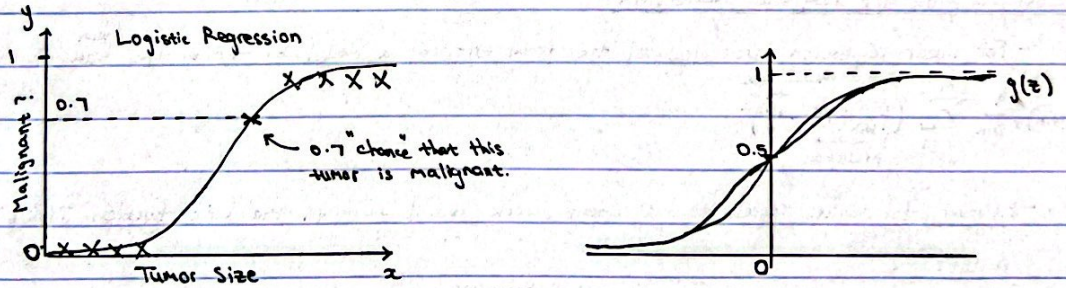
- We want to classify things into specific categories.
- Binary Classification: Output y is either true or false.
- Linear regression doesn't work:



Logistic Regression

- We can better fit the data using a sigmoid function:

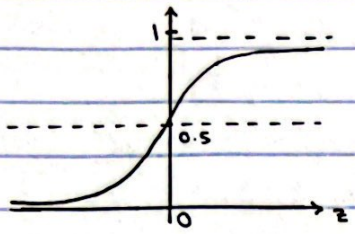
$$g(z) = \frac{1}{1+e^{-z}}, \quad 0 \leq g(z) \leq 1$$



- The value of $z = f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$.
- The output of the logistic regression function is the "probability" that the input is 1.

Decision Boundary

- Previously, we used logistic regression to graph a best-fit line to our classification points. To determine which points lie in which category, we use a decision boundary to separate our points.



- Is $f_{\vec{w},b}(\vec{x}) \geq 0.5$ (threshold) — Yes: $\hat{y}=1$; no: $\hat{y}=0$

$\because f_{\vec{w},b}(\vec{x}) = g(\vec{w}\cdot\vec{x}+b) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$,

$f_{\vec{w},b}(\vec{x}) \geq 0.5$ when $g(z) \geq 0.5$

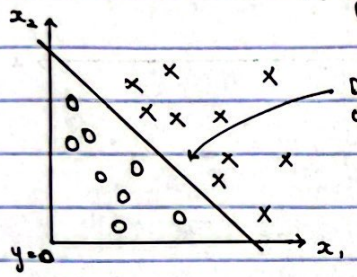
$z \geq 0$

$\therefore \hat{y}=1$ when $\vec{w}\cdot\vec{x}+b \geq 0$

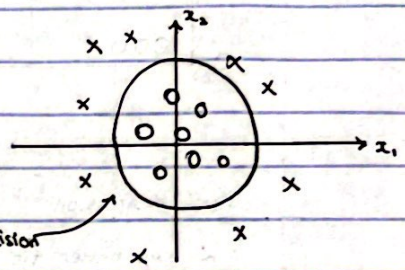
$\hat{y}=0$ when $\vec{w}\cdot\vec{x}+b < 0$

- Let the decision boundary be when $z = \vec{w}\cdot\vec{x}+b = 0$

(Eqn. for decision boundary when we solve)



Decision boundary separates/ classifies our points.



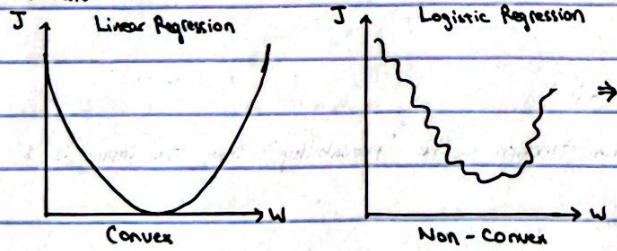
Non-linear decision boundaries.

Cost Function for Logistic Regression

- For linear regression, we calculated the total error of a model by finding the mean of the squared error function

$$J(\vec{w},b) = \frac{1}{2m}\sum_{i=1}^{m}\left(\underbrace{f_{\vec{w},b}(\vec{x}^{(i)})}_{\text{prediction}} - \underbrace{\vec{y}^{(i)}}_{\text{Actual Value}}\right)^2$$

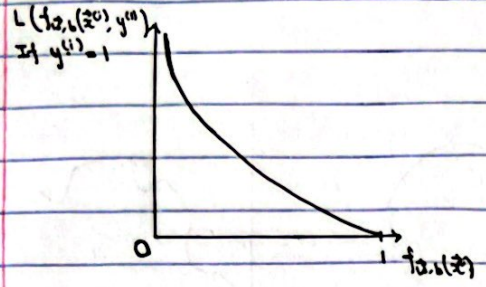
- However, for logistic regression this doesn't work, and if we graph the cost function $J(\vec{w},b)$, we get a non-convex result:



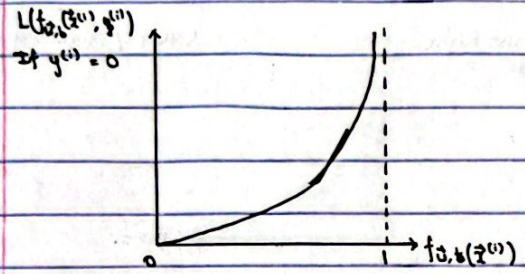
⇒ Unable to use gradient descent

- Loss Function: Function we used to calculate the cost for each training point in logistic regression:

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)}=1 \\ -\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)}=0 \end{cases}$$



- When the true value, $y^{(i)} = 1$, if our prediction is 1, we get a loss of 0.
- However, if our prediction $\to 0$ our loss $\to \infty$.



- On the other hand, when $y^{(i)} = 0$, and our prediction is 0, our loss function returns 0.
- Conversely as our prediction $f_{\vec{w},b}(\vec{x}^{(i)}) \to 1$, loss $\to \infty$.

- We can calculate the total cost of our model:

$$J(\vec{w},b) = \frac{1}{m}\sum_{i=0}^{m} \frac{1}{2} L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right)$$

Simplified Cost Function

- Because our y values are only 1 and 0, we can simplify our loss function to:

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = -y^{(i)} \log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) - (1-y^{(i)}) \log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right)$$

Gradient Descent Implementation

- Implementing gradient descent for logistic regression is the same for linear regression.

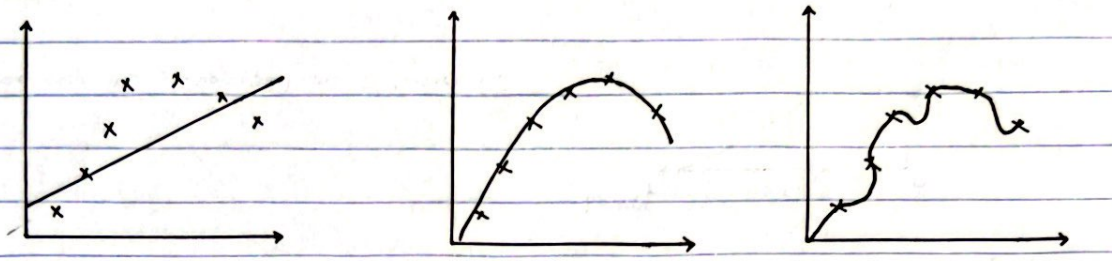
$$\text{cost: } J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)}) \log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right)\right]$$

$$\text{repeat } \{\ w_j = w_j - \alpha \frac{d}{dw_j} J(\vec{w},b),$$
$$b = b - \alpha \frac{d}{db} J(\vec{w},b)\ \}$$

$$\text{repeat } \left\{ w_j = w_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{n}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right) x_j^{(i)}\right]\right.$$
$$\Rightarrow$$
$$\left. b = b - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)\right]\right\}$$

- Same concepts apply:
  - Monitor gradient descent to ensure it converges.
  - Vectorization to make it run faster.
  - Feature scaling.

The Problem of Overfitting



Underfit / High Bias | "Just Right" | Overfit / High Variance

Addressing Overfitting

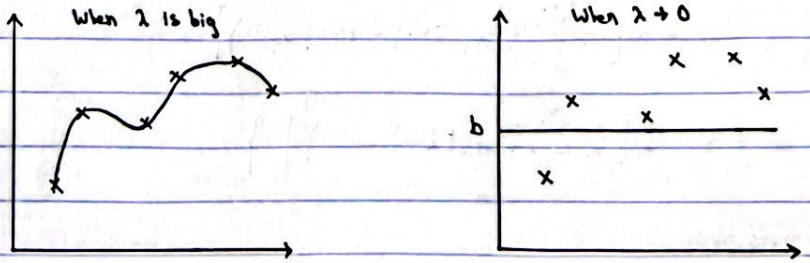
1. Add more training data.
2. Reduce features / Feature selection.
3. Regularization : Instead of entirely removing features, reduce the size of parameters.

Cost Function with Regularization

- Instead of eliminating features to prevent overfitting, we can use regularization to "punish" the model for placing too much emphasis on any particular feature.
- The modified cost function is :

$$J(\vec{w}, b) = \underbrace{\frac{1}{2m} \sum_{i=1}^{m} \left[f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right]^2}_{\text{Mean Squared Error}} + \underbrace{\frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2}_{\substack{\text{Regularization} \\ \text{Term}}}$$

- So now, when we try to minimize for $J(\vec{w}, b)$, we have to balance between finding the minimum mean squared error and regularization term.
  - If the regularization parameter λ is small, $\frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2 \to 0$ and we end up overfitting.
  - If λ is large, to find the minimum $J(\vec{w}, b)$, the only way is to make it such that all values of w are close to 0, where $f_{\vec{w},b}(\vec{x}) = \overset{\approx 0}{w_1} x + \overset{\approx 0}{w_2} x + \dots + b$



Regularized Linear Regression

- With regularization our new gradient descent becomes:

repeat { $w_j = w_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left[\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right) x_j^{(i)} \right] + \frac{\lambda}{m} w_j \right]$

$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left[f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)} \right]$

}

Regularized Logistic Regression

- Likewise, we can penalize overly large features by adding a regularization term to the cost function:

$$J(\vec{w},b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(f_{\vec{w},b}(\vec{x}^{(i)}) \right) + (1 - y^{(i)}) \log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)}) \right) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$$

Repeat { $w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w},b)$,

$b = b - \alpha \frac{\partial}{\partial b} J(\vec{w},b)$ } (same as linear regression!)