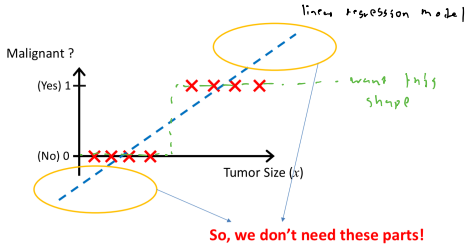
By turning classes into #'s can use linear regression

e.g. Sunny = 0, rainy = 1

* note Logistic regression is a classifier

Logistic Regression Classifier with One Feature

- we know that for our classifier, the **output should be either 1 or 0**!

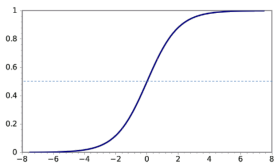


So we use Sigmoid

Sigmoid Function

- Sigmoid Function (Logistic Function):

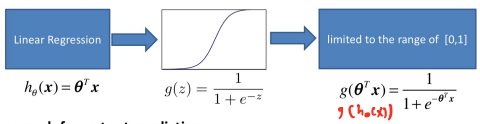
$$g(z) = \frac{1}{1+e^{-z}}$$



limits range to [0,1]

Logistic Regression Model

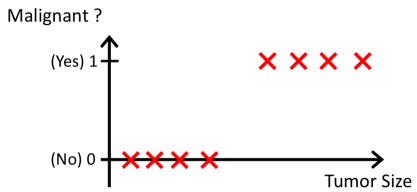
- Using Sigmoid Function (Logistic Function):



New approach for output prediction:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

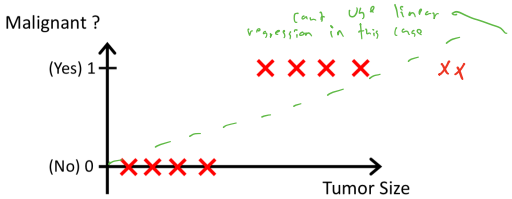
So, Now the NEW $h_\theta(x)$ is limited to the range of [0,1].



- New approach: $h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$

- can output between 0 & 1
- to generate binary output, can compare with a threshold

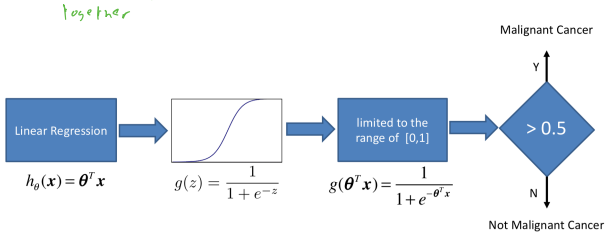
i.e. predict "y = 1" if $h_\theta(x) \geq 0.5$
predict "y = 0" if $h_\theta(x) < 0.5$



- New approach: $h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$

After limiting the range and discretizing

have to integrate together



How to Select the Parameters?

Training set (m training samples):

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$$

Feature Vector: $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ $\quad x_0 = 1, y \in \{0, 1\}$

Gradient Descent for Logistic Regression

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \qquad x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

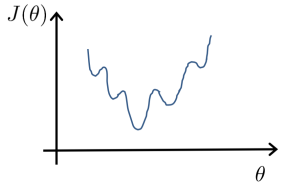
- **Previous definition of Cost function for linear regression:**

$$J(\theta) = J(\theta_0, \theta_1, \theta_2, \ldots, \theta_n) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Previous definition of Cost function for linear regression:

$$J(\theta) = J(\theta_0, \theta_1, \theta_2, \ldots, \theta_n) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$



This Cost Function is not Convex for Logistic Regression (It has many local minimums)! → gradient descent might not find global min. for error

use log function to define an alternative convex cost function

why it works

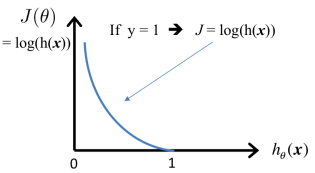
New Cost function:

$$J(\theta) = J(\theta_0, \theta_1, \ldots, \theta_n)$$

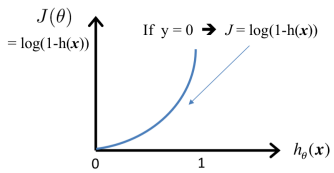
$$= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right]$$

want to find parameters θ s.t. $\min_\theta J(\theta)$

Note: $y = 0$ or 1 always



- when y=1, and h(x) is close to 1, the cost will be very small, and when y=1, and h(x) is close to 0, the cost will be too high.



- when y=0, and h(x) is close to 0, the cost will be very small, and when y=0, and h(x) is close to 1, the cost will be too high.

Gradient Descent for Multiple Variables

- Output (probability): $h_\theta(\boldsymbol{x}) = \dfrac{1}{1+e^{-\theta^T x}}$
- Cost function: $J(\boldsymbol{\theta}) = -\dfrac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h_\theta(\boldsymbol{x}^{(i)}) + (1-y^{(i)})\log(1-h_\theta(\boldsymbol{x}^{(i)}))\right]$
- **Gradient descent:**

Repeat { $\theta_j := \theta_j - \alpha\dfrac{\partial}{\partial\theta_j}J(\theta_0,\ldots,\theta_n)$

$= \theta_j - \alpha\dfrac{1}{m}\sum_{i=1}^{m}(h_\theta(\boldsymbol{x}^{(i)}) - y^{(i)})\,x_j^{(i)}$

} (simultaneously update for every $j = 0,\ldots,n$)

- **Algorithm looks identical to linear regression (except for the definition of $h(x)$)!!!**

$\theta^T x$

Evaluating accuracy of our predictive model

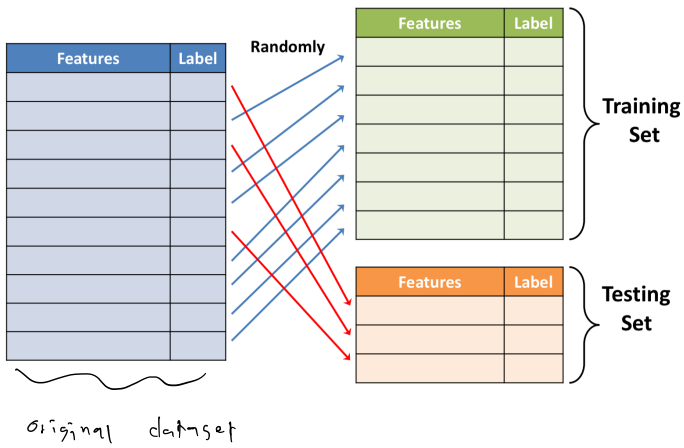
Here is a simple way to evaluate the accuracy of our predictive model:

1- Let's split the dataset **RANDOMLY** into two new datasets: **Training Set** (e.g. 70% of the data samples) and **Testing Set** (30% of the data).

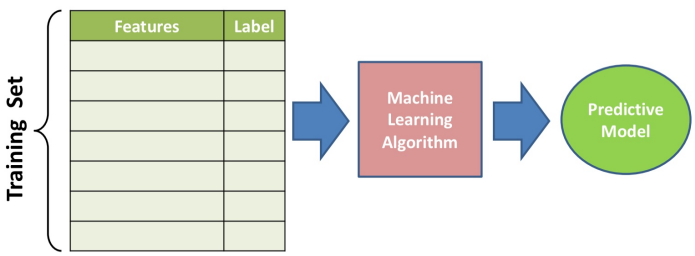
2- Let's **pretend** that we do **NOT** know the label of the Testing Set!

3- Let's Train the model **ONLY on Training Set**, and then Predict on the Testing Set!

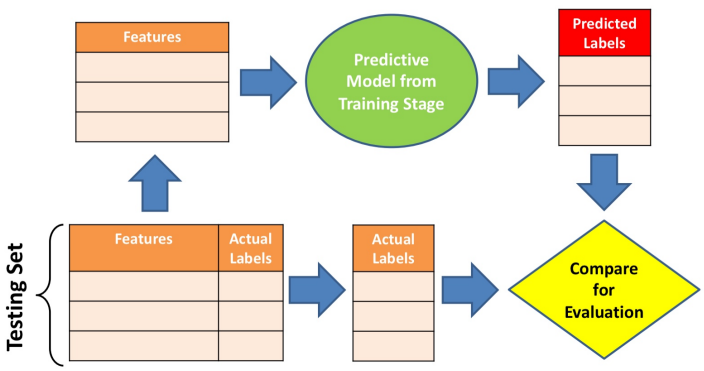
4- After prediction, we can compare the **predicted labels** for the Testing Set with the **actual labels** of it to evaluate the accuracy of our prediction!



Training Stage



Testing Stage



Cross Validation

- We saw how to split the dataset into Training and Testing sets, Fit the model on "training set", and then predict on "testing set" to evaluate the accuracy.
- The problem with this method is that **the results may depend on the choice of split**. For example, if you are lucky, some easily predictable samples may happen to be located in the testing set (or vice versa!).
- In order to get fair results, we can repeat the splitting process several times, compute the prediction accuracy for each split, and then average the results.
- **Cross Validation tries to repeat the splitting procedure K times in a smart way such that all data samples will be used in "testing set" one time and in "Training Set" (K-1) times!**

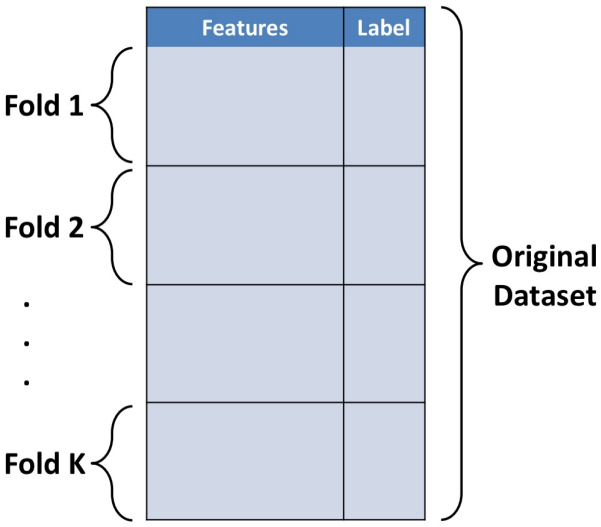
Cross Validation

Three main steps for K-fold cross-validation:

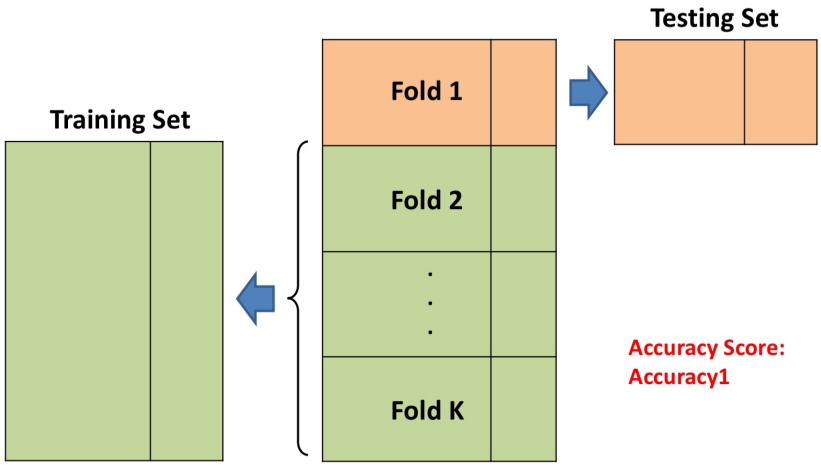
1. Partition the dataset Randomly into K equal, non-overlapping sections (called Fold).
2. Use one of the sections as **testing set** at a time and the union of the other (K-1) sections as the **training set**. Perform training stage, testing stage, and compute the accuracy based on the split each time. Repeat this procedure K times, so that each one of the K sections is used as **testing set** one time, and as a part of **training set** (K-1) times.
3. Calculate the average of the accuracies as final result.

Note: K is arbitrary, but Using K=10 (10-fold cross-validation) is very common and recommended in machine learning.

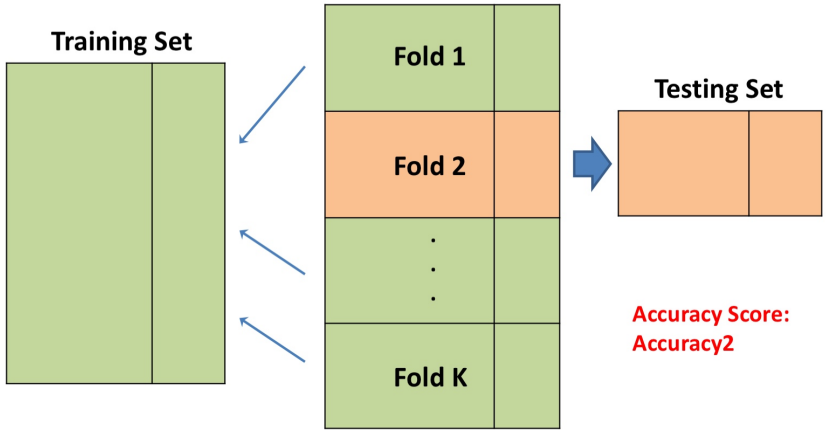
Cross Validation



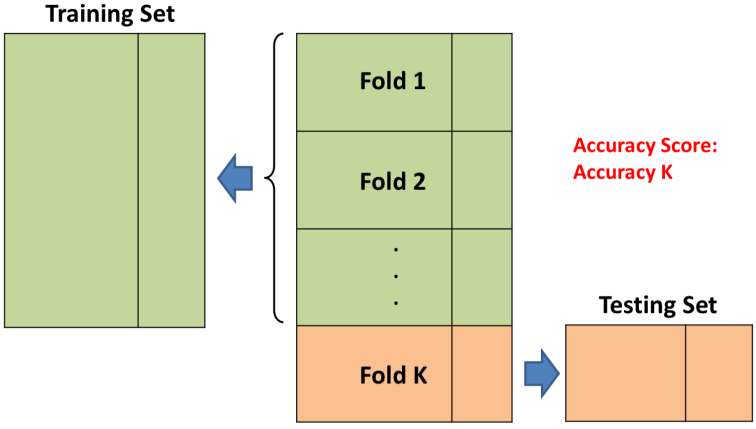
Cross Validation – Round 1



Cross Validation – Round 2



Cross Validation – Round K



Cross Validation

Accuracy_Score_Total =

(Accuracy 1 + Accuracy 2 + Accuracy 3 + … + Accuracy K) / K