Lesson 2: Linear Regression

Mean vs Total Squared (or Absolute) Error

Q: How do we know if we should use the mean or the total squared (or absolute) error?

The total squared error is the sum of errors at each point, given by the following equation

$$M = \sum_{i=1}^{m} \frac{1}{2}(y - \hat{y})^2,$$

whereas the mean squared error is the average of these errors, given by the equation, where m is the number of points:

$$T = \sum_{i=1}^{m} \frac{1}{2m}(y - \hat{y})^2.$$

So, it doesn't really matters. As we can see, the total squared error is just a multiple of the mean squared error, since

$$M = mT.$$

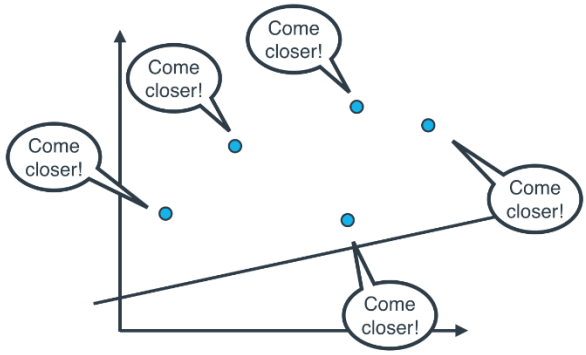
Therefore, since derivatives are linear functions, the gradient of T is also mm times the gradient of M. The gradient descent step consists of subtracting the gradient of the error times the learning rate α. Therefore, choosing between the mean squared error and the total squared error just amounts to picking a different learning rate.

Therefore, if we use the mean error or the total error, the algorithm will just end up picking a different learning rate.

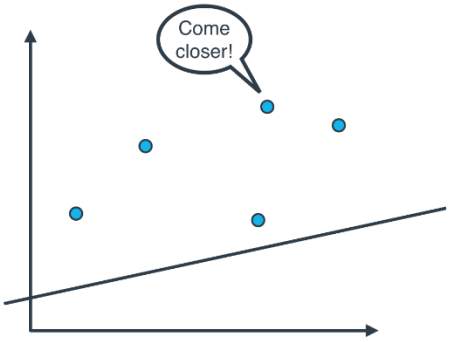
Batch vs Stochastic Gradient Descent

we've seen two ways of doing linear regression. By **applying the squared (or absolute) trick at every point in our data**

- one by one and repeating this process many times.
- all at the same time and repeating this process many times.



Batch



Stochastic

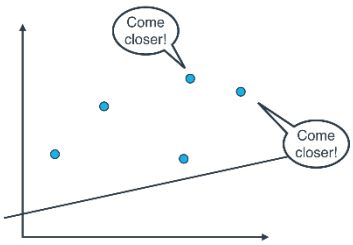
Batch Gradient Descent:

- We consider the whole batch and update the weights based on these.

Stochastic Gradient Descent:

- We consider the single point and update the weights based on it one by one.

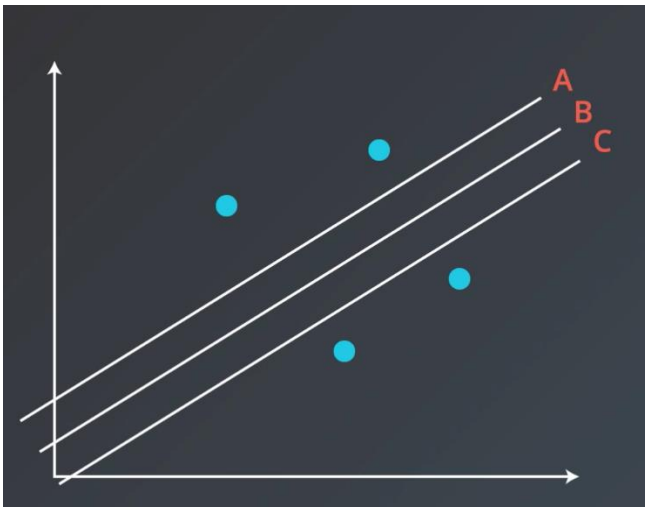
Mini Batch gradient Descent: Batch GD + Stochastic GD



Mini-batch

If your data is huge, both BGD and SGD are a bit slow, computationally. So we use Mini Batch GD

Absolute error vs Squared error



Q. Which of the three lines gives you a smaller Mean Absolute Error?

All of them give the same error.

Q. Which of the three lines gives you a smaller Mean Square Error?

B

Linear Regression in Scikit learning

We use scikit-learn's *LinearRegression* class. This class provides the function ***fit()*** to fit the model to your data.

```
>>> from sklearn.linear_model import LinearRegression
>>> model = LinearRegression()
>>> model.fit(x_values, y_values)
```

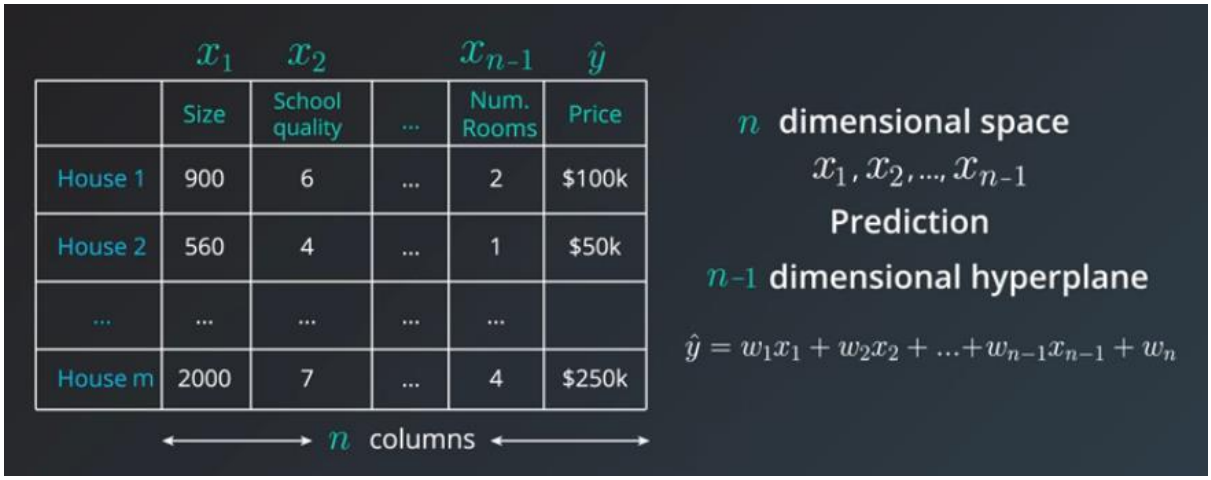
the ***model*** variable is a linear regression model that has been fitted to the data ***x_values*** and ***y_values***. Fitting the model means finding the best line that fits the training data.

```
>>> print(model.predict([ [127], [248] ]))
[[ 438.94308857, 127.14839521]]
```

The reason for predicting on an array like [127] and not just 127, is because you can have a model that makes a prediction using multiple features.

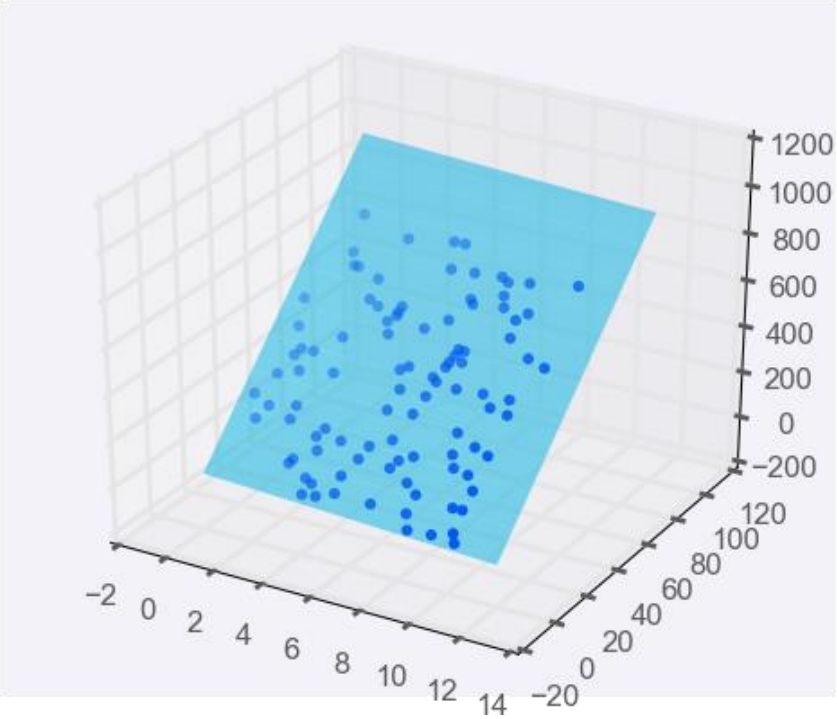
Higher dimensions

- Suppose of Housing Prices data, we have n columns, i.e. n-1 columns as input and 1 as output.
- Prediction is made in n-1 dimensional hyperplane.



Multiple Linear Regression

- Dependent variables : features
- Independent variables: target



$$y = m_1x_1 + m_2x_2 + b$$

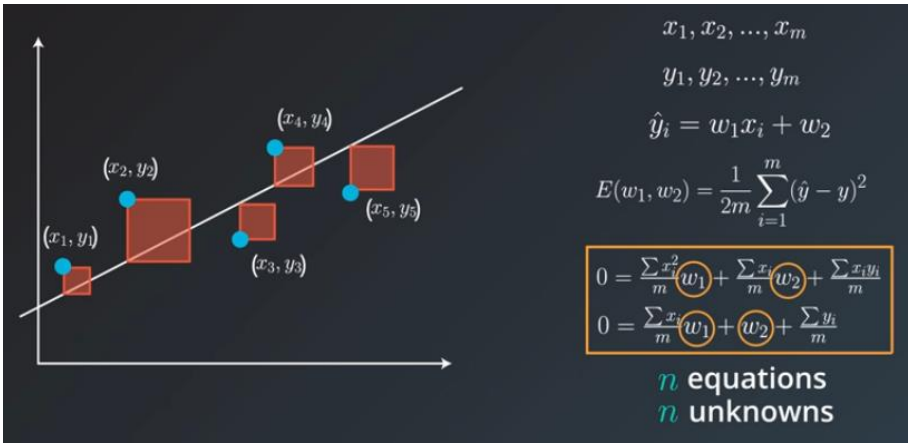
When there's just one predictor, the linear regression model is a line, but as you add more predictor variables, you're adding more dimensions to the picture.

Adding a predictor variable to go to two predictor variables means that the predicting equation is:

Closed Form Solution

Since there are huge number of points n, we get n equations with n unknowns. And calculations get expensive. Also, we will have to invert the matrix of size n*n.

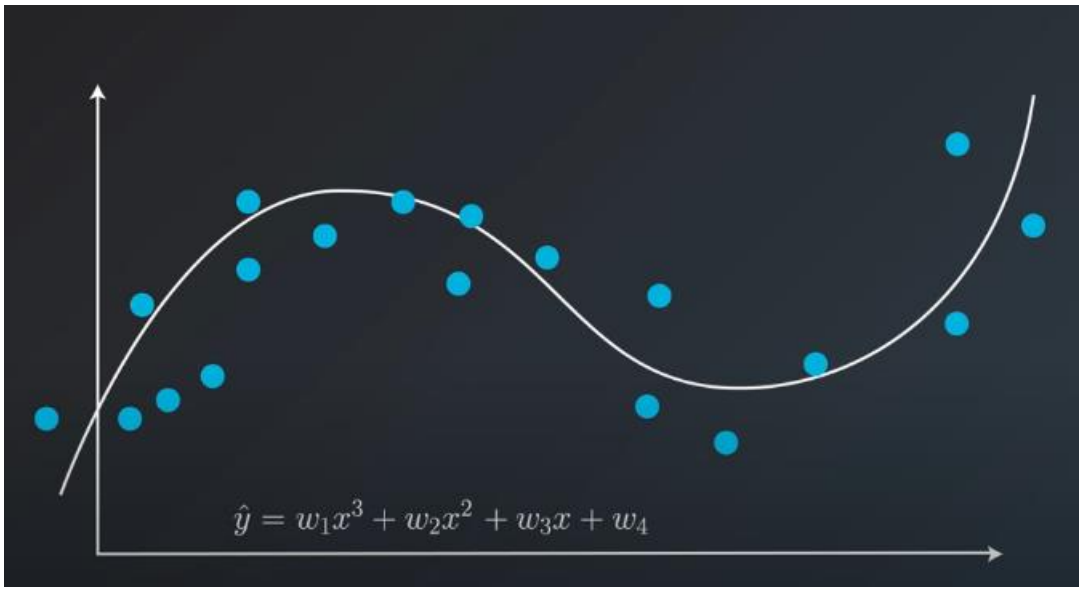
So we use gradient descent ~ pretty close to what we need.



Linear Regression Warnings:

- Linear Regression Works Best When the Data is Linear
- Linear Regression is Sensitive to Outliers

Polynomial Regression

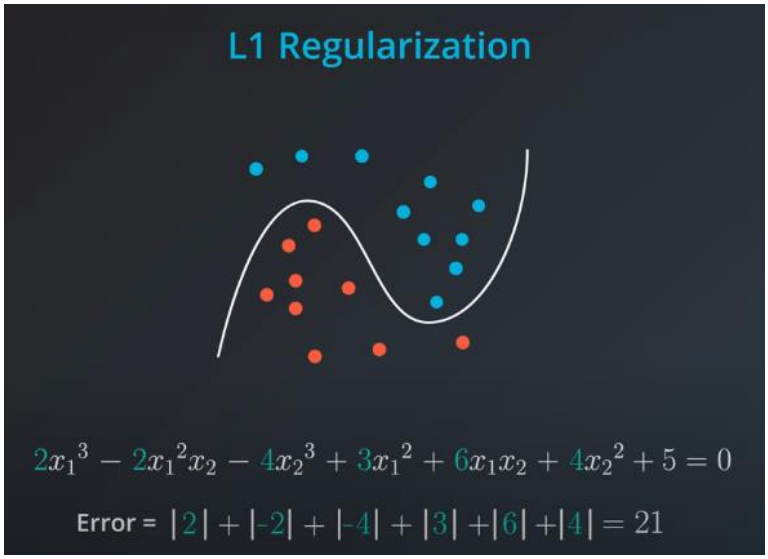


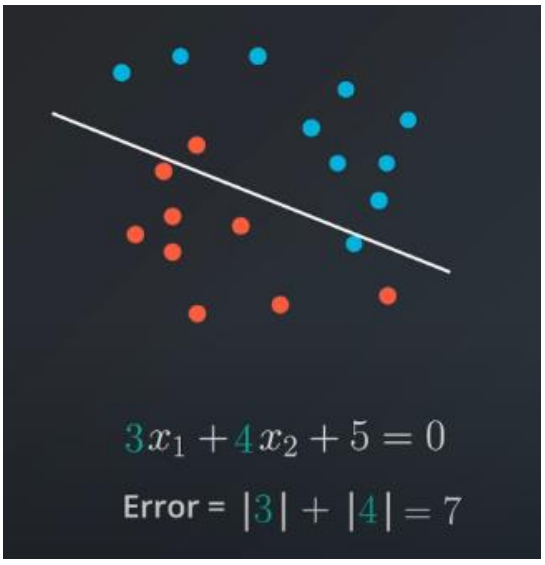

```
# TODO: Add import statements
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Assign the data to predictor and outcome variables
# TODO: Load the data
train_data = pd.read_csv('data.csv')
X = train_data['Var_X'].values.reshape(-1, 1)
y = train_data['Var_Y'].values

# Create polynomial features
# TODO: Create a PolynomialFeatures object, then fit and transform the
# predictor feature
poly_feat = PolynomialFeatures(degree = 4)
X_poly = poly_feat.fit_transform(X)

# Make and fit the polynomial regression model
# TODO: Create a LinearRegression object and fit it to the polynomial predictor
# features
poly_model = LinearRegression(fit_intercept = False).fit(X_poly, y)
```

Regularization

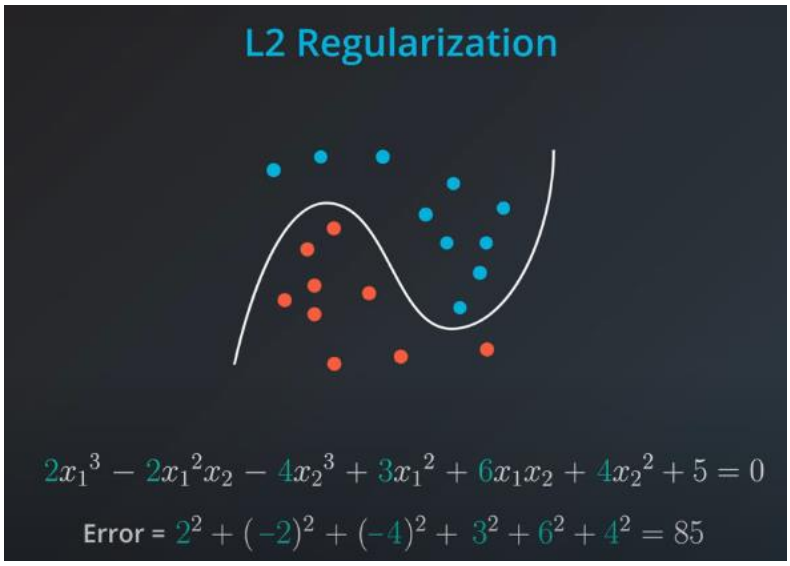
Since complex models have more coefficients, the error calculation wrt each variable lead to higher errors. Whereas the simpler models have better generalization.

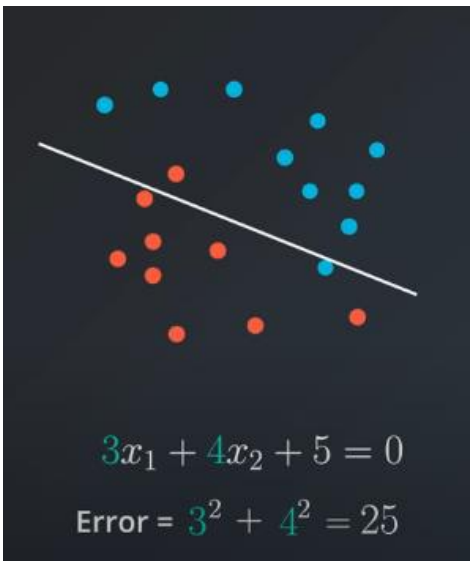
L1 Regularization: taking the coefficients and considering them for error calculations.



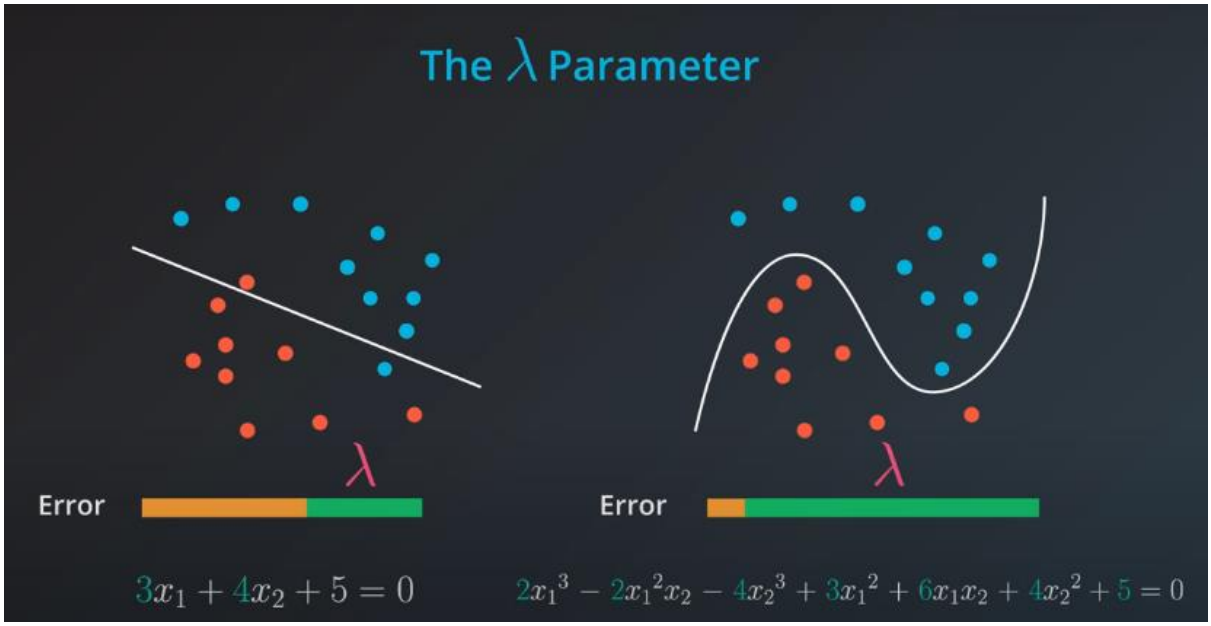


L2 Regularization: instead of absolute we add square of the coefficients.

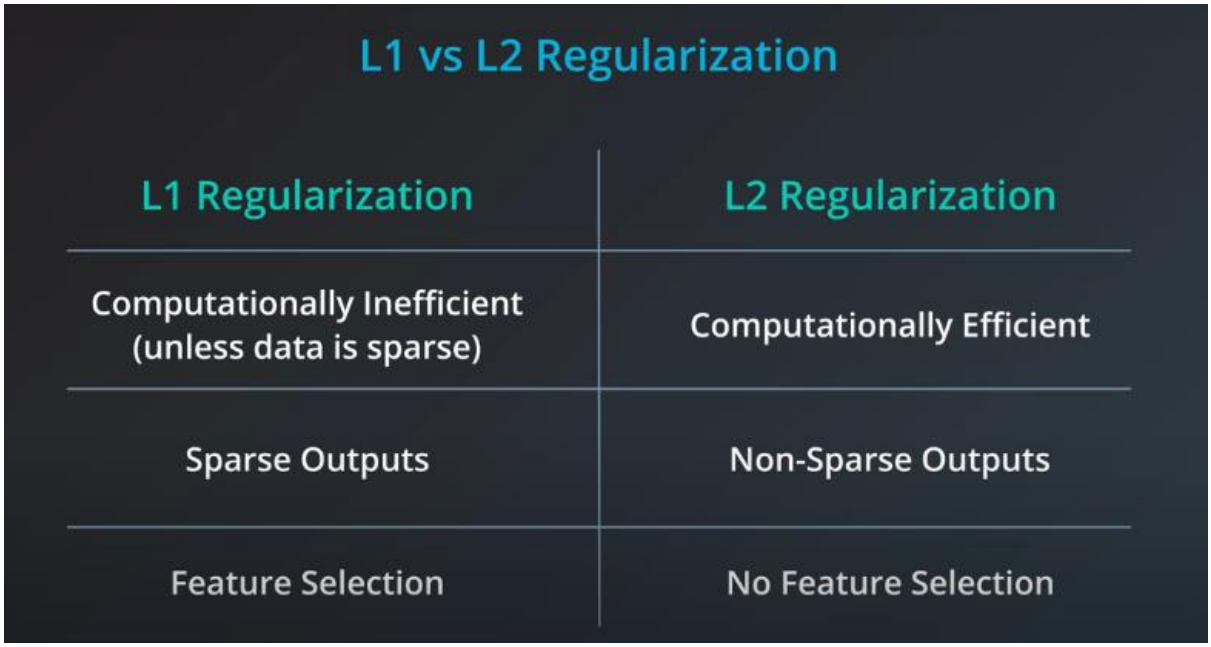




We can see, complex models gets punished a lot as compared to the simpler models.
So, we use a new parameter, lamda:



Tip: Use small lamda for complex regularization, whereas large lambda for simpler regularization.




```
# TODO: Add import statements
import numpy as np
import pandas as pd
from sklearn.linear_model import Lasso

# Assign the data to predictor and outcome variables
# TODO: Load the data
train_data = pd.read_csv('data.csv', header = None)
X = train_data.iloc[:,:-1]
```

```
y = train_data.iloc[:,-1]

# TODO: Create the linear regression model with lasso regularization.
lasso_reg = Lasso()

# TODO: Fit the model.
lasso_reg.fit(X, y)

# TODO: Retrieve and print out the coefficients from the regression model.
reg_coef = lasso_reg.coef_
print(reg_coef)
```

Feature Scaling

way of transforming your data into a common range of values. There are two common scalings:

- **Standardizing:** taking each value of your column, subtracting the mean of the column, and then dividing by the standard deviation of the column.
  - `df["height_standard"] = (df["height"] - df["height"].mean()) / df["height"].std()`

- **Normalizing:** data are scaled between 0 and 1.
  - `df["height_normal"] = (df["height"] - df["height"].min()) / \`
  - `(df["height"].max() - df['height'].min())`

When Should I Use Feature Scaling?

In many machine learning algorithms, the result will change depending on the units of your data.
This is especially true in two specific cases:

1. When your algorithm uses a distance-based metric to predict.
2. When you incorporate regularization.

Distance Based Metrics

supervised learning technique that is based on the distance points are from one another called
Support Vector Machines (or SVMs).
Another technique that involves distance based methods to determine a prediction is *k-nearest*
neighbors (or k-nn). With either of these techniques, choosing not to scale your data may lead to
drastically different (and likely misleading) ending predictions.

Regularization: LINK

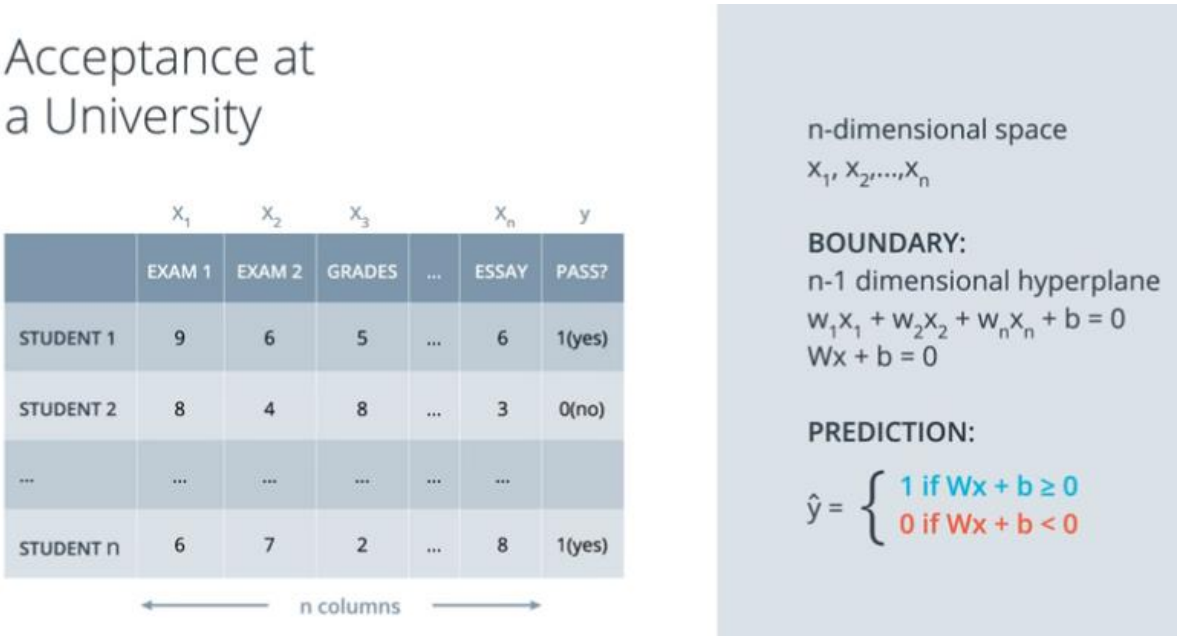
- When you start introducing regularization, you will again want to scale the features of your model. The penalty on particular coefficients in regularized linear regression techniques depends largely on the scale associated with the features. When one feature is on a small range, say from 0 to 10, and another is on a large range, say from 0 to 1 000 000, applying regularization is going to unfairly punish the feature with the small range. Features with small ranges need to have larger coefficients compared to features with large ranges in order to have the same effect on the outcome of the data. (Think about how ab=ba for two numbers aa and bb.) Therefore, if regularization could remove one of those two features with the same net increase in error, it would rather remove the small-ranged feature with the large coefficient, since that would reduce the regularization term the most.
- feature scaling can speed up convergence of your machine learning algorithms, which is an important consideration when you scale machine learning applications.

```
# TODO: Add import statements
import numpy as np
import pandas as pd
from sklearn.linear_model import Lasso
from sklearn.preprocessing import StandardScaler

# Assign the data to predictor and outcome variables
# TODO: Load the data
train_data = pd.read_csv('data.csv', header = None)
X = train_data.iloc[:,:-1]
y = train_data.iloc[:,-1]

# TODO: Create the standardization scaling object.
scaler = StandardScaler()

# TODO: Fit the standardization parameters and scale the data.
X_scaled = scaler.fit_transform(X)

# TODO: Create the linear regression model with lasso regularization.
lasso_reg = Lasso()

# TODO: Fit the model.
lasso_reg.fit(X_scaled, y)

# TODO: Retrieve and print out the coefficients from the regression model.
reg_coef = lasso_reg.coef_
print(reg_coef)
```

RECAP

- **Gradient descent** as a method to optimize your linear models.
- **Multiple Linear Regression** as a technique for when you are comparing more than two variables.
- **Polynomial Regression** for relationships between variables that aren't linear.
- **Regularization** as a technique to assure that your models will not only fit to the data available, but also extend to new situations.

Lesson 3: Perceptron Algorithm



What would the dimensions be for input features (x), the weights (W), and the bias (b) to satisfy (Wx + b)?

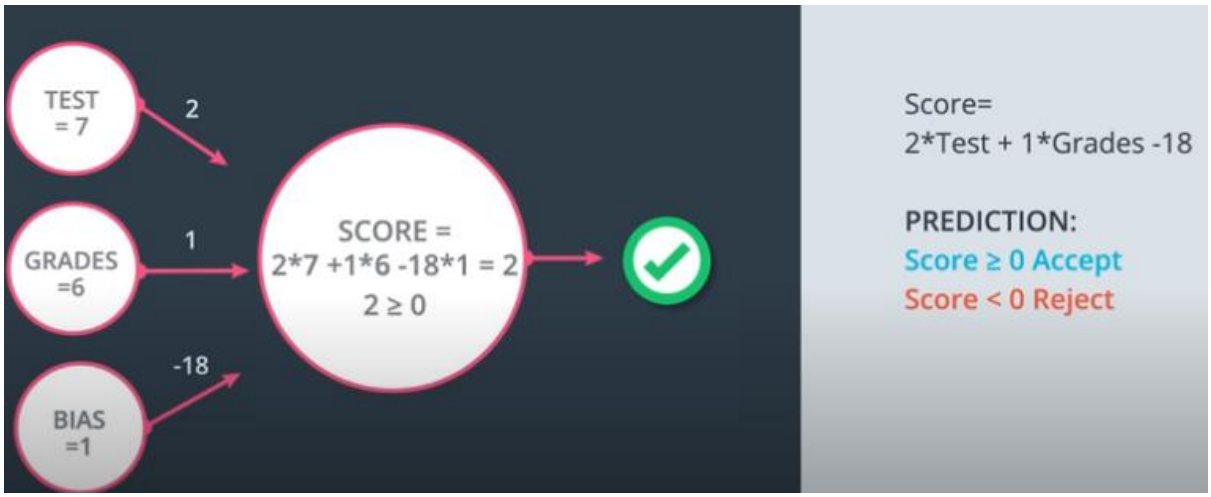
W : (1*n)

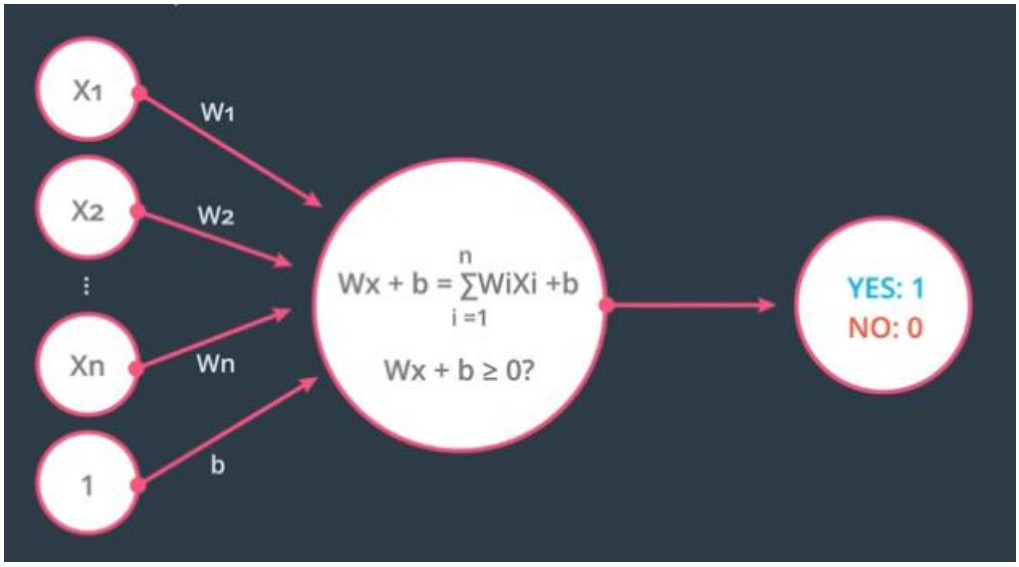
X : (n*1)

b : (1*1)

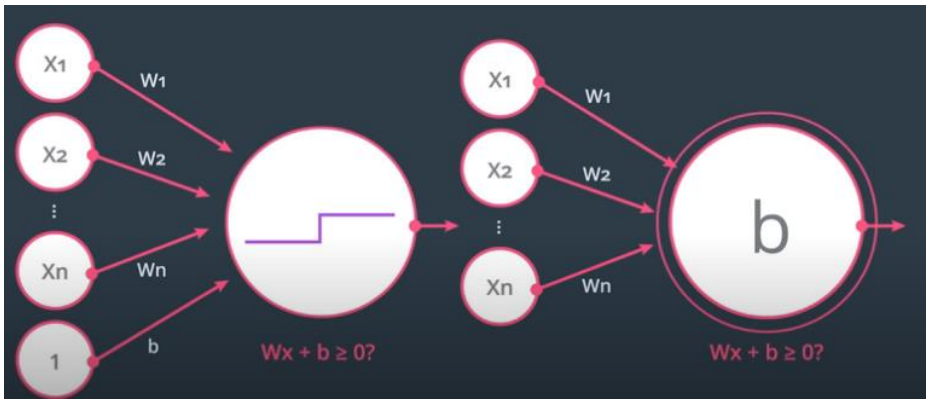
Perceptron

Representing the above problem in terms of perceptron.



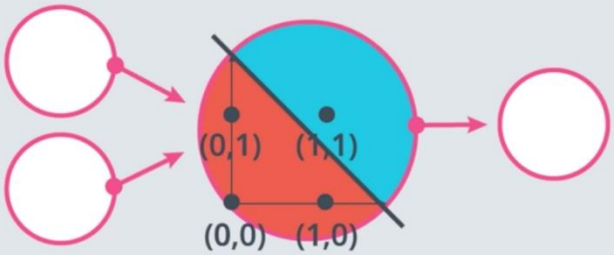


The bias can be represented as an extra node, or else it can be added in the next stage:



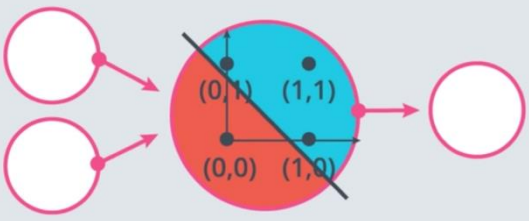
Perceptron as Logical Operators

- Perceptron model for AND w1,w2,b =[1,1,-2]



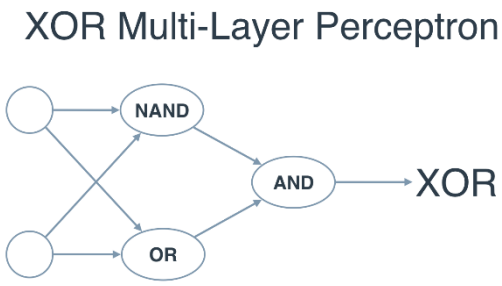
IN	IN	OUT
1	1	1
1	0	0
0	1	0
0	0	0

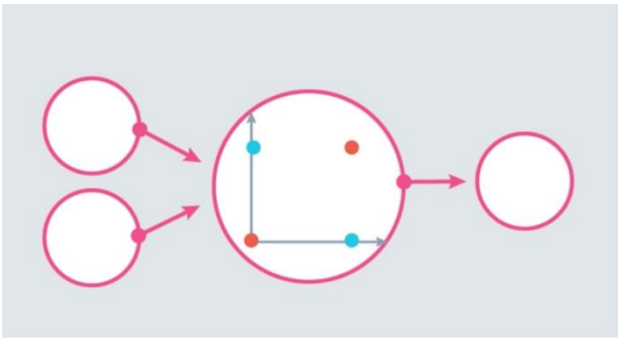
- Perceptron model for OR



IN	IN	OUT
1	1	1
1	0	1
0	1	1
0	0	0

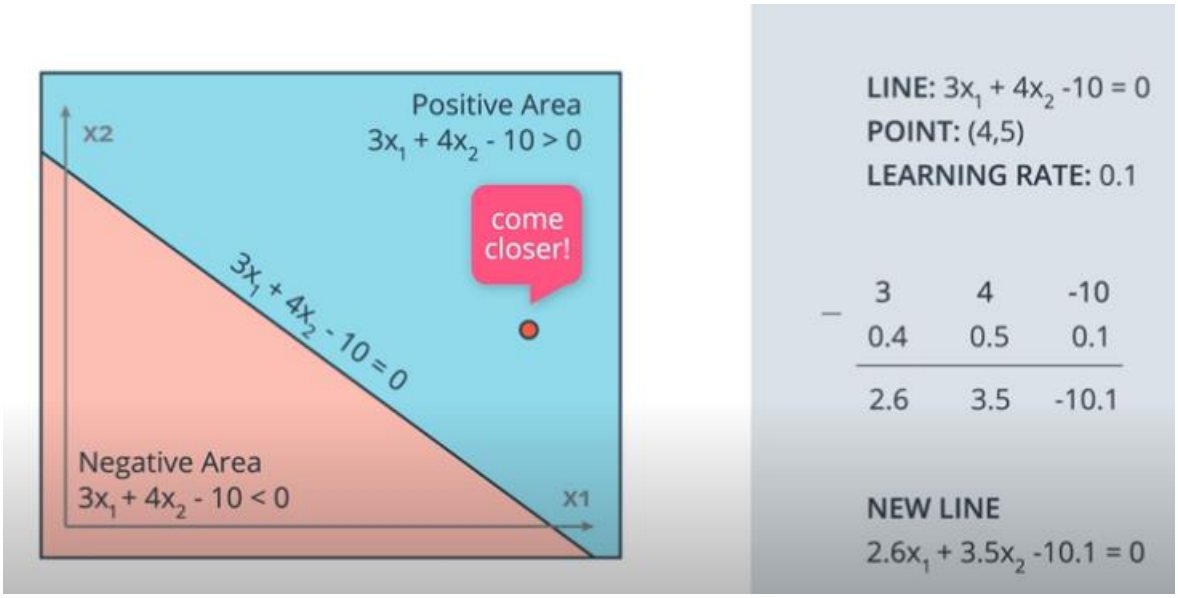
- What are two ways to go from an AND perceptron to an OR perceptron?
    - Increase the weights.
    - Decrease the magnitude of the bias.
- **NOT perceptron: [w1,w2,b] = [0,-1, 0.5]**
  doesn't care about w1, needs -ve w2, and +ve but smaller bias.
- **XOR:** if we introduce the NAND operator as the combination of AND and NOT, then we get the following two-layer perceptron that will model XOR. That's our first neural network!





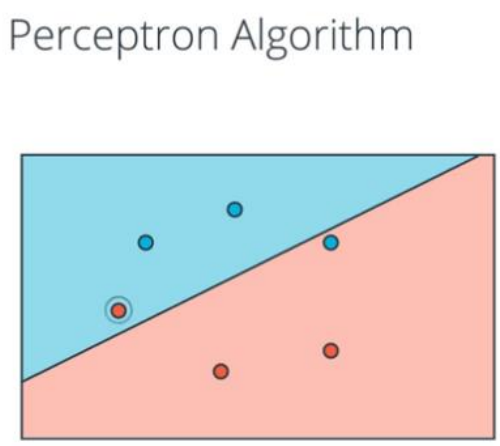
IN	IN	OUT
1	1	0
1	0	1
0	1	1
0	0	0

How do we move the lines towards the points?
3,4,-1 are coefficients in equation.
0.4, 0.5 and 0.1 come for the point (4,5) and 1 is takes as general term. We multiply these by LR.



When we choose LR = 0.1, it takes 10 small steps to reach the points.

Perceptron Algorithms



1. Start with random weights: $w_1, \ldots, w_n, b$

2. For every misclassified point $(x_1,\ldots,x_n)$:
   2.1. If prediction = 0:
   - For i = 1 ...n
   - Change $w_i + \alpha x_i$
   - Change b to b + α

   2.2. If prediction = 1:
   - For i = 1 ...n
   - Change $w_i - \alpha x_i$
   - Change b to b - α

Note:
We change $W_i = W_i + \alpha x_i$

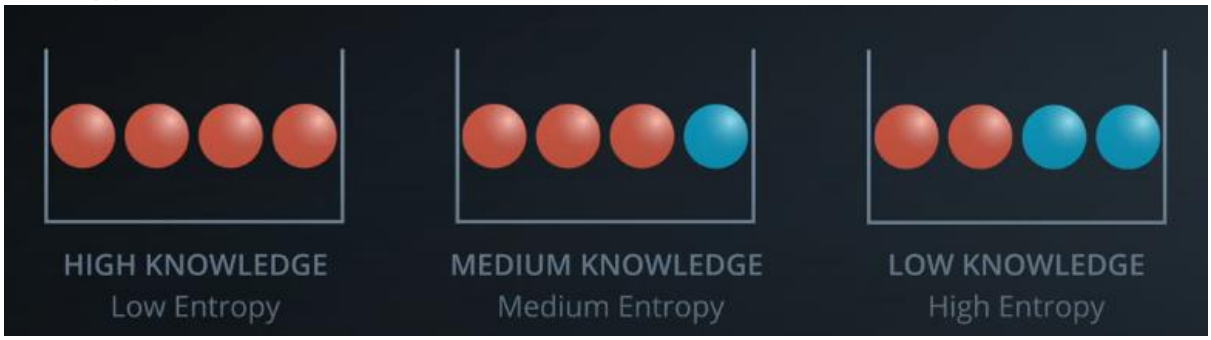
(+- depends upon situation)

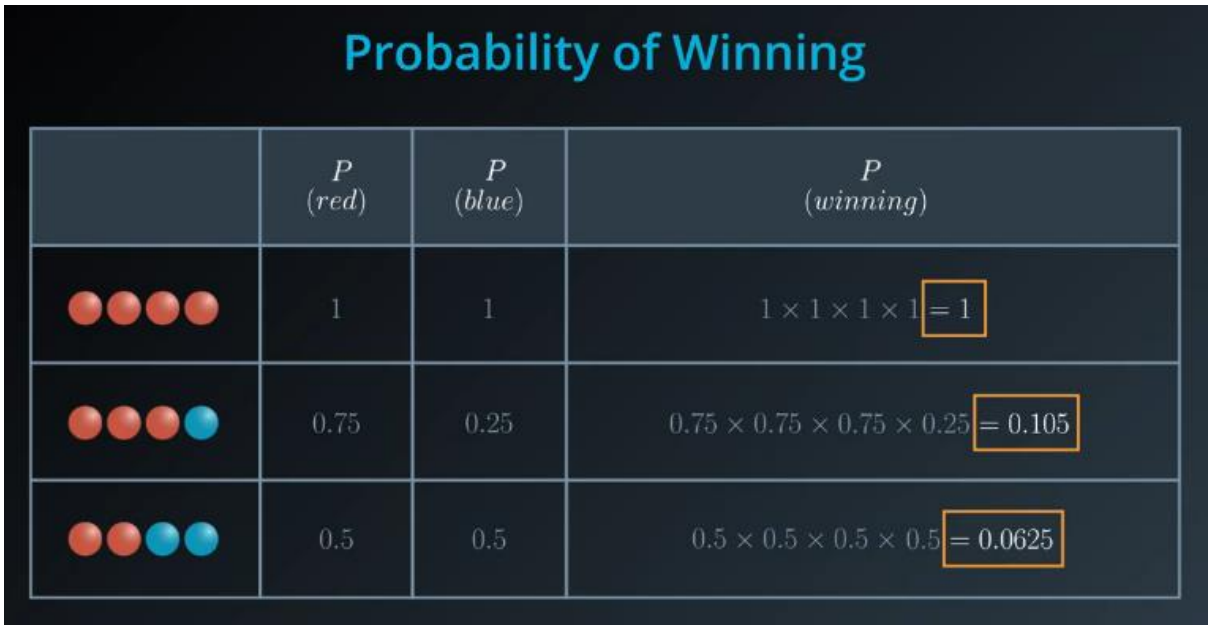
Pseudocode:

- If the point is correctly classified, do nothing.
- If the point is classified positive, but it has a negative label, subtract αp, αq, and α from w1, w2& b respectively.
- If the point is classified negative, but it has a positive label, add αp,αq, and α to w1, w2, & b respectively.

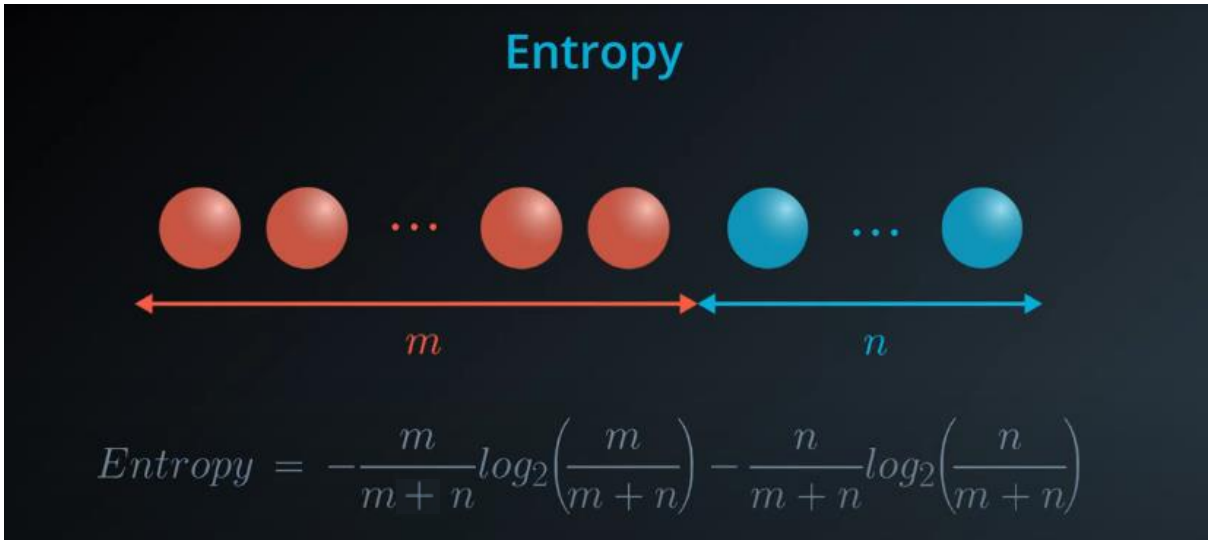
Decision Trees

Entropy





Instead of taking the products, we take the sum using *log*.



Multi-class Entropy

- equation for entropy for a bucket with m red balls and n blue balls:

$$entropy = -\frac{m}{m+n}\log_2(\frac{m}{m+n}) - \frac{n}{m+n}\log_2(\frac{n}{m+n})$$

- We can state this in terms of probabilities instead for the number of red balls as p1 & p2

$$p_1 = \frac{m}{m+n} \qquad p_2 = \frac{n}{m+n}$$

$$entropy = -p_1\log_2(p_1) - p_2\log_2(p_2)$$

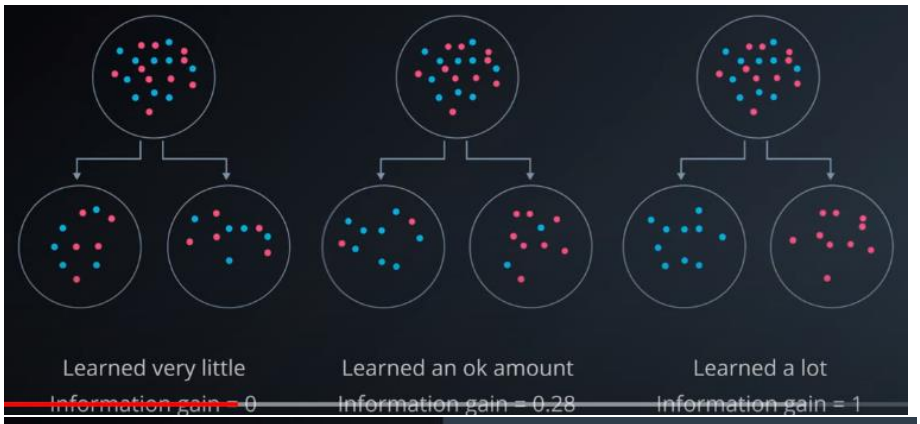
We can extend this equation for more than one class:

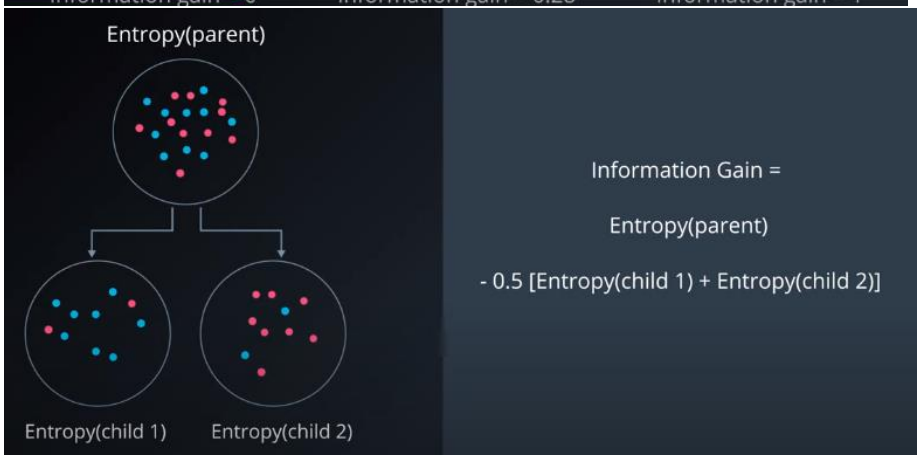
$$entropy = -p_1\log_2(p_1) - p_2\log_2(p_2) - \ldots - p_n\log_2(p_n) = -\sum_{i=1}^{n} p_i\log_2(p_i)$$

So, when all the elements are same: **minEntropy** = 0
maxEntropy increases with the number of possible outcomes.

Example: 8R, 3B, 2Y balls. Total entropy: 1.334

Information Gain

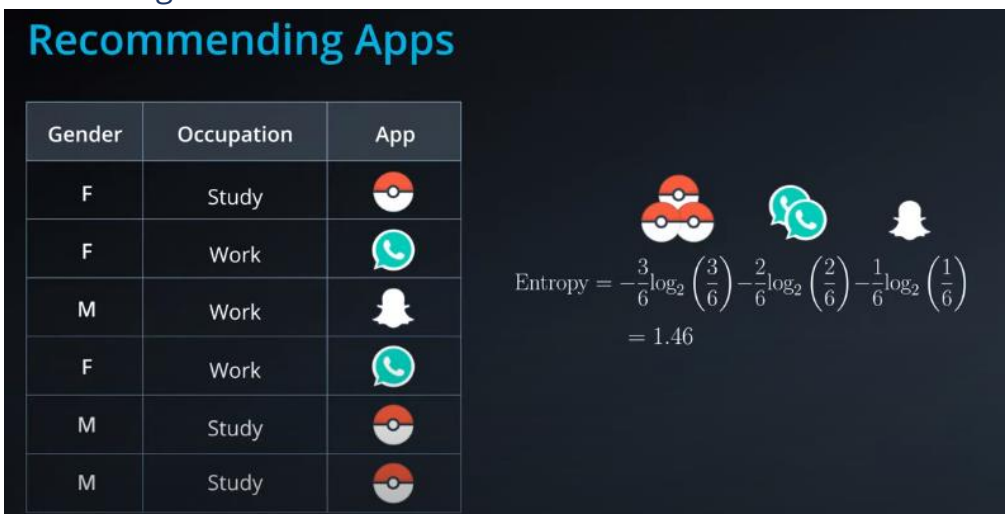


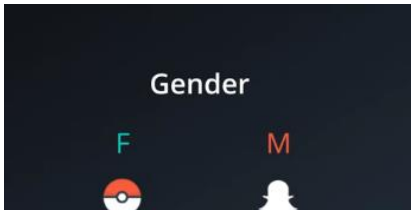


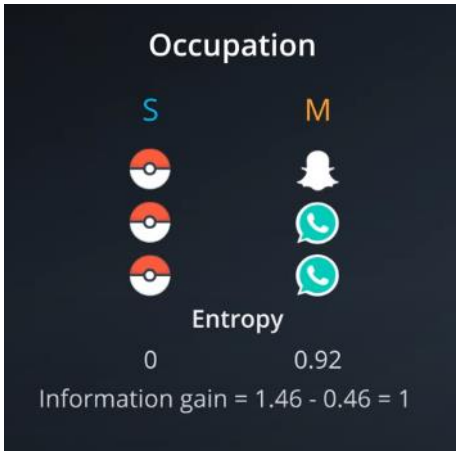
- The child groups are weighted equally in this case since they're both the same size, for all splits.
- In general, average entropy for the child groups will need to be a weighted average, based on the number of cases in each child group. That is, for mm items in the first child group and nn items in the second child group, the information gain is:

$$\text{Information Gain} = \text{Entropy}(Parent) - \left[\frac{m}{m+n}\text{Entropy}(Child_1) + \frac{n}{m+n}\text{Entropy}(Child_2)\right]$$

Maximizing the IG







We observe that splitting by

occupation gives more information gain. So, heres the final one:

How to calculate the IG code:

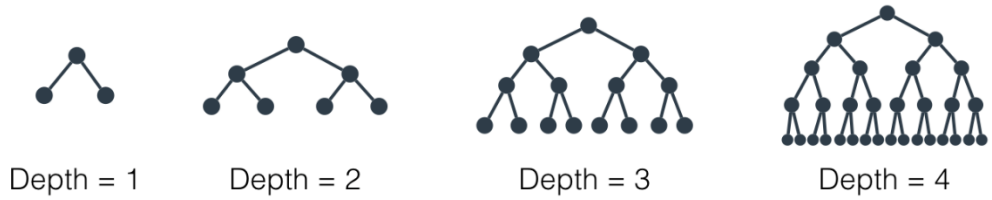
```
def two_group_ent(first, tot):          #calculates entropy for 2 classes
    return -(first/tot*np.log2(first/tot) +
            (tot-first)/tot*np.log2((tot-first)/tot))
tot_ent = two_group_ent(10, 24)
g17_ent = 15/24 * two_group_ent(11,15) + 9/24 * two_group_ent(6,9)
answer = tot_ent - g17_ent
```

	A	B	C
1	**Species**	**Color**	**Length (mm)**
2	Mobug	Brown	11.6
3	Mobug	Blue	16.3
4	Lobug	Blue	15.1
5	Lobug	Green	23.7
6	Lobug	Blue	18.4
7	Lobug	Brown	17.1
8	Mobug	Brown	15.7
9	Lobug	Green	18.6
10	Lobug	Blue	22.9
11	Lobug	Blue	21
12	Lobug	Blue	20.5
13	Mobug	Green	21.2
14	Mobug	Brown	13.8
15	Lobug	Blue	14.5
16	Lobug	Green	24.8
17	Mobug	Brown	18.2
18	Lobug	Green	17.9
19	Lobug	Green	22.7
20	Mobug	Green	19.9
21	Mobug	Blue	14.6
22	Mobug	Blue	19.2
23	Lobug	Brown	14.1
24	Lobug	Green	18.8
25	Mobug	Blue	13.1
26			

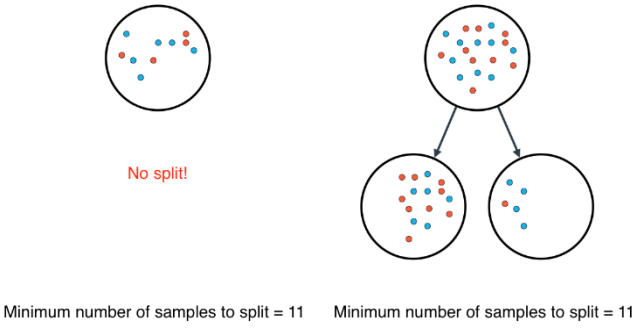
Hyperparameters for Decision Trees

In order to create decision trees that will generalize to new problems well, we can tune a number of different aspects about the trees. We call the different aspects of a decision tree "hyperparameters". These are some of the most important hyperparameters used in decision trees:

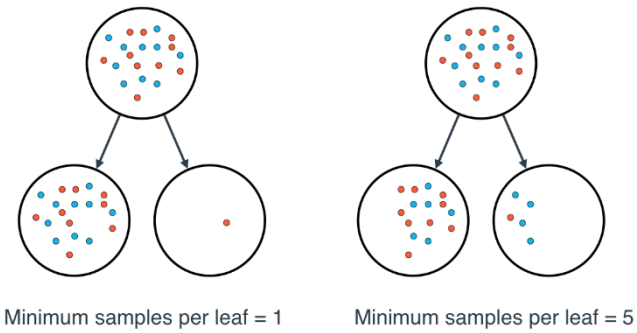
- **Maximum Depth:** the largest possible length between the root to a leaf. A tree of maximum length k can have at most 2^k leaves.



- **Minimum number of samples to split:** A node must have at least `min_samples_split` samples in order to be large enough to split. If a node has fewer samples than `min_samples_split` samples, it will not be split, and the splitting process stops.
  - However, `min_samples_split` doesn't control the minimum size of leaves.



- **Minimum number of samples per leaf**
  When integer: its min samples/leaf.
  When float: its in %. 0.1 ~ 10%



FEATURE	UNDERFITTING/OVERFITTING
Small maximum depth	Underfitting
Large maximum depth	Overfitting
Small minimum samples per split	Overfitting
Large minimum samples per split	Underfitting

- Large depth very often causes overfitting, since a tree that is too deep, can memorize the data. Small depth can result in a very simple model, which may cause underfitting.
- Small minimum samples per split may result in a complicated, highly branched tree, which can mean the model has memorized the data, or in other words, overfit. Large minimum samples may result in the tree not having enough flexibility to get built, and may result in underfitting.

Decision Tree in sklearn

For your decision tree model, you'll be using scikit-learn's `Decision Tree Classifier` class.

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> model = DecisionTreeClassifier()
>>> model.fit(x_values, y_values)
```

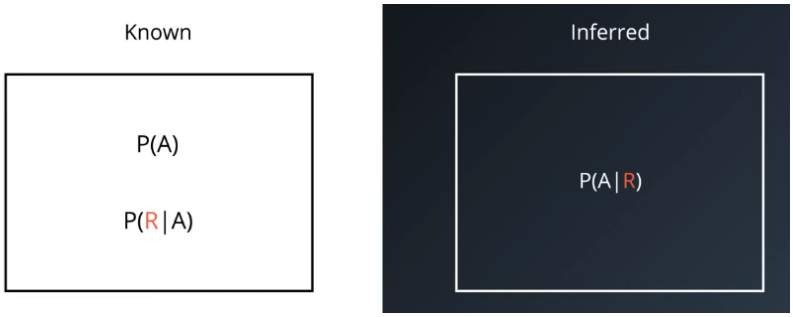
Hyperparameters

In practice, the most common ones are

- `max_depth`: The maximum number of levels in the tree.
- `min_samples_leaf`: The minimum number of samples allowed in a leaf.
- `min_samples_split`: The minimum number of samples required to split an internal node.

Naïve Bayes

$$P(A|R) = \frac{P(A)P(R|A)}{P(A)P(R|A) + P(B)P(R|B)}$$



$$P(A)\,P(R|A)$$

$$P(B)\,P(R|B)$$

BAYES THEOREM

$$P(S|+) = \frac{P(S)P(+|S)}{P(S)P(+|S) + P(H)P(+|H)}$$

$$= \frac{0.0001 * 0.99}{0.0001 * 0.99 + 0.9999 * 0.01}$$

$$= 0.0098$$

$$< 1\%$$

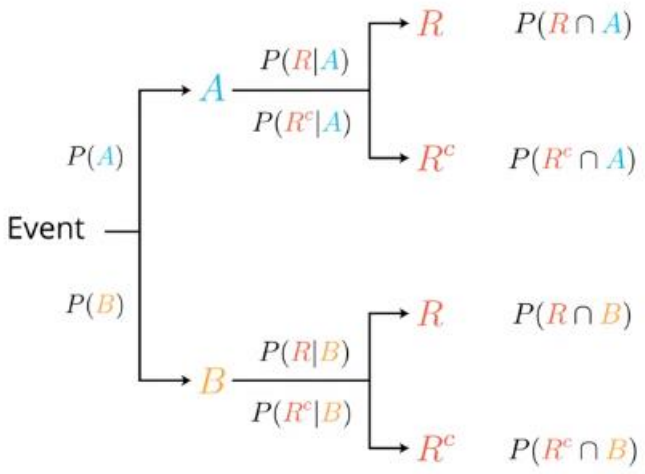
S: sick
H: healthy
+ : positive

$P(S) = 0.0001$

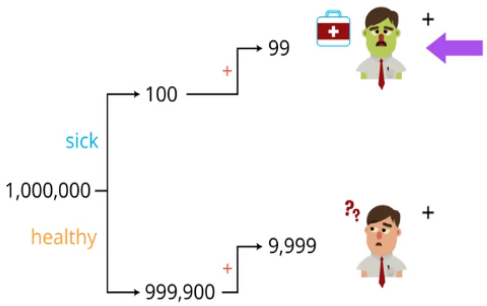
$P(H) = 0.9999$

$P(+|S) = 0.99$

$P(+|H) = 0.01$



Example: when person +ve, what are the chances he is really sick. 1 out of 10000 patient is sick and test has accuracy of 99%.



1 out of every 10,000 patients is sick

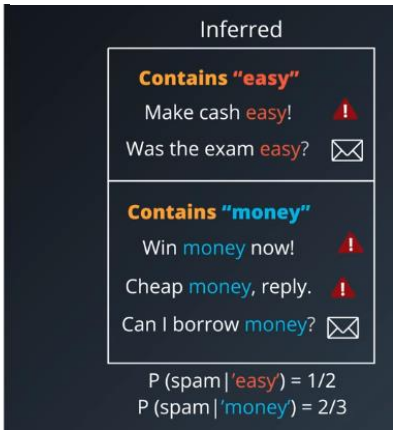
Test has 99% accuracy

Patient tested positive

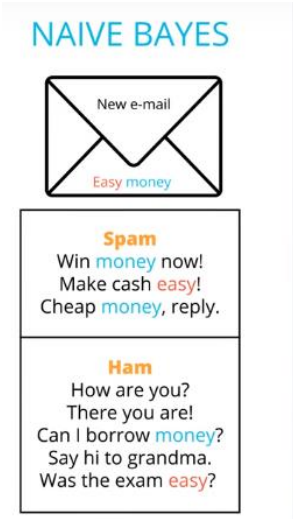
$$P(\text{sick}|+) = \frac{99}{9,999 + 99}$$

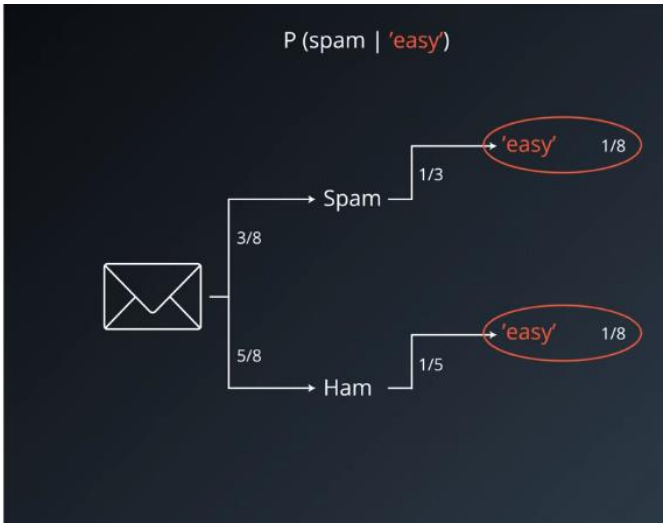
$$= 0.0098$$

$$< 1$$



Below, since both have same probability when we normalize it becomes 50%.





Conditional Probability

$$P(\text{spam} \mid \text{'easy', 'money'}) \propto P(\text{'easy', 'money'} \mid \text{spam})\, P(\text{spam})$$

$$P(A \mid B) \propto P(B \mid A) P(A)$$

CONDITIONAL PROBABILITY

$$P(\text{spam} \mid \text{'easy'}, \text{'money'}) \propto P(\text{'easy'} \mid \text{spam})\, P(\text{'money'} \mid \text{spam})\, P(\text{spam})$$

$$P(A \,\&\, B) = P(A)P(B)$$

NAIVE ASSUMPTION

So when we check the mails and fill in the values:

$$P(\text{spam} \mid \text{'easy'}, \text{'money'}) \propto \underset{1/12}{P(\text{'easy'} \mid \text{spam})}\, \underset{1/3}{P(\text{'money'} \mid \text{spam})}\, \underset{3/8}{P(\text{spam})}$$

$$P(\text{ham} \mid \text{'easy'}, \text{'money'}) \propto \underset{1/40}{P(\text{'easy'} \mid \text{ham})}\, \underset{1/5}{P(\text{'money'} \mid \text{ham})}\, \underset{5/8}{P(\text{ham})}$$

These are the proportional probabilities: which results in following values:

$$P(\text{spam} \mid \text{'easy'}, \text{'money'}) \propto \frac{1}{12} \qquad \frac{\frac{1}{12}}{\frac{1}{12} + \frac{1}{40}}$$

$$P(\text{ham} \mid \text{'easy'}, \text{'money'}) \propto \frac{1}{40} \qquad \frac{\frac{1}{40}}{\frac{1}{12} + \frac{1}{40}}$$

Example: Suppose you have a bag with three standard 6-sided dice with face values [1,2,3,4,5,6] and two non-standard 6-sided dice with face values [2,3,3,4,4,5]. Someone draws a die from the bag, rolls it, and announces it was a 3. What is the probability that the die that was rolled was a standard die?

Solution:

Probability of 3 from whole bag = p(s) * p(3/s) +p(ns) *p(3/ns)

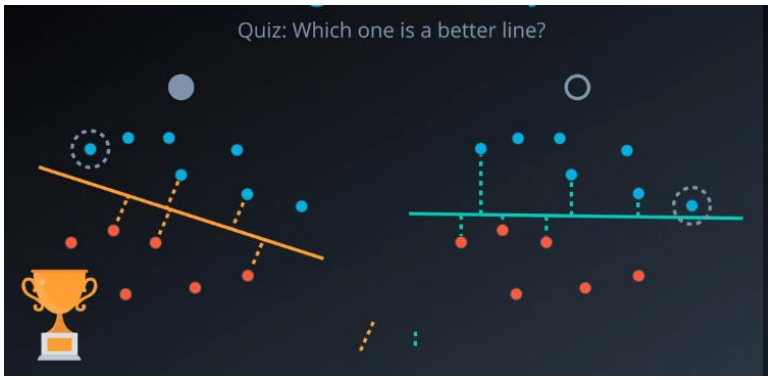
(1/10) / (1/10 + 2/15) = 3/7 = 0.429

SPAM CLASSIFIER:

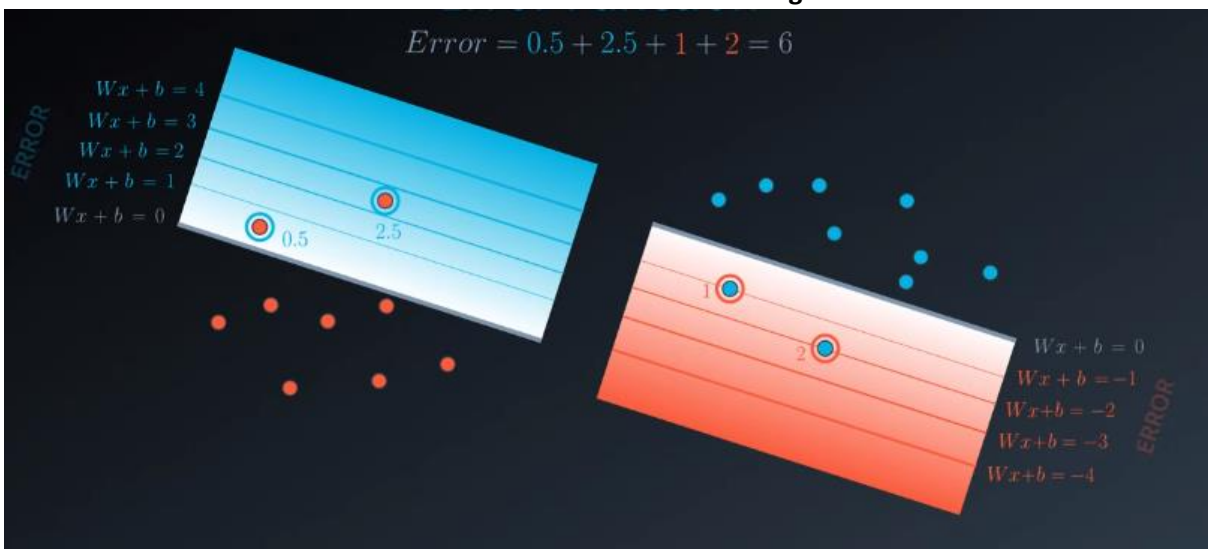
LINK to NLP- Udacity AI nanodegree program.

Support Vector Machine

General concept: We compare the minimum distance of a point from the line and check which one them is smallest. And it wins!

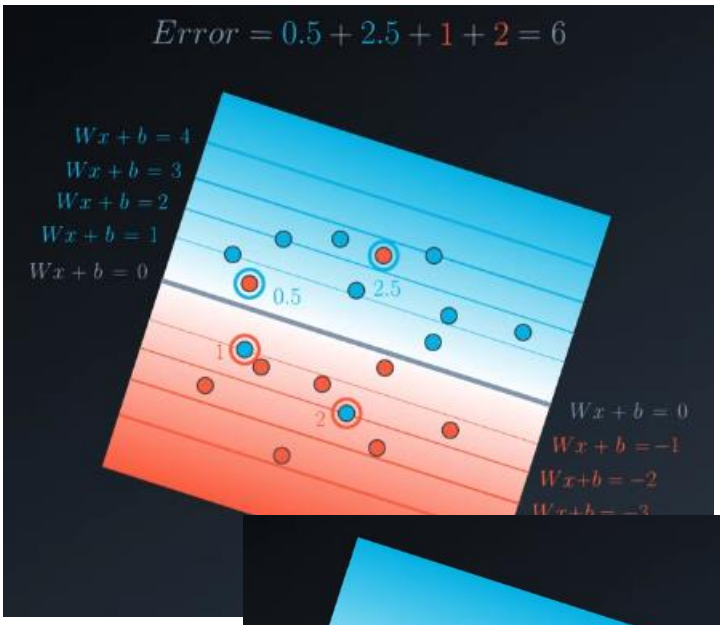


- Lets try to solve this using **Perceptron Algorithm:**
- In this case the new error will be sum of **classification error + margin error**

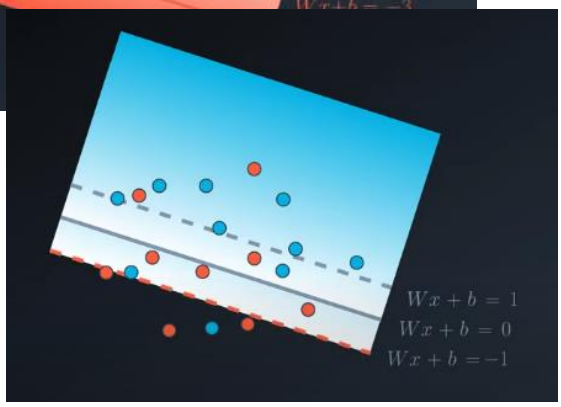


- The points away from the line get punishes more. The gradient shows the magnitude of errors the points will be punished.

- When combined all the points look like this:
- Here the motive will be find appropriate W and b. This is the perceptron algo.



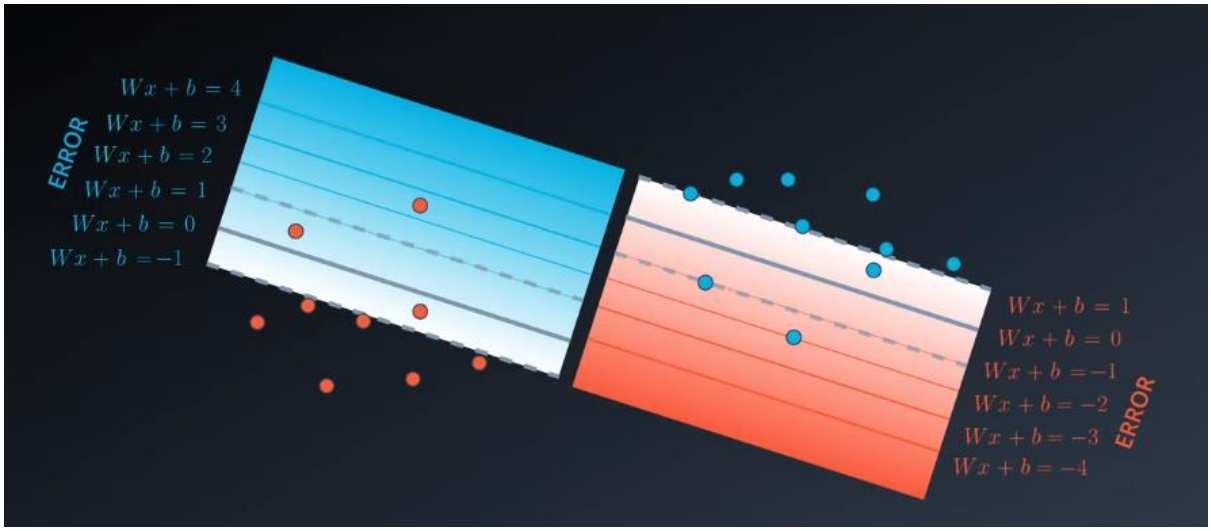
Lets try to solve the same problem using **SVM** now:

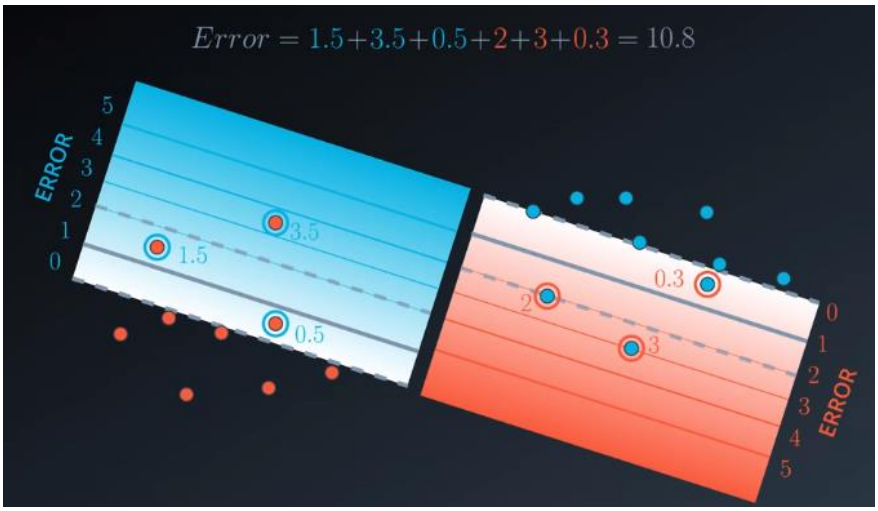


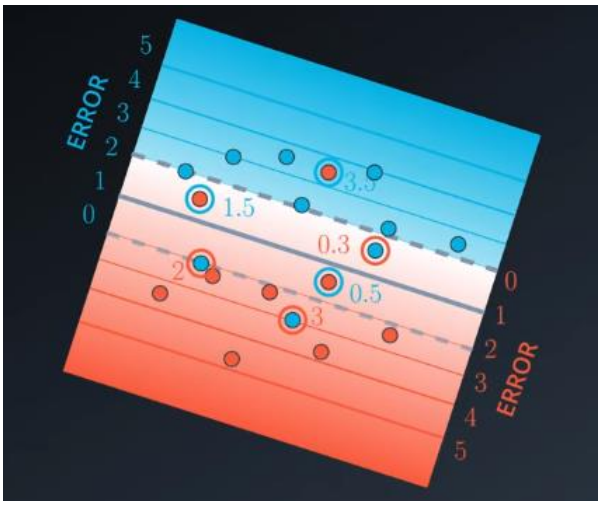
Classification Error

This time when calculating the error we will consider the blue line starting from the bottom, whereas last time we started from the line itself. Here we punish more, and punishment starts very early.

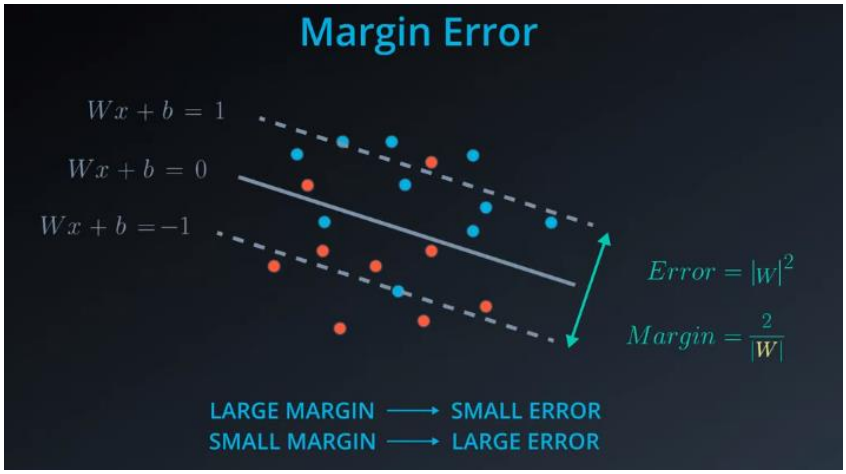
And the red error starts from the blue dotted line.



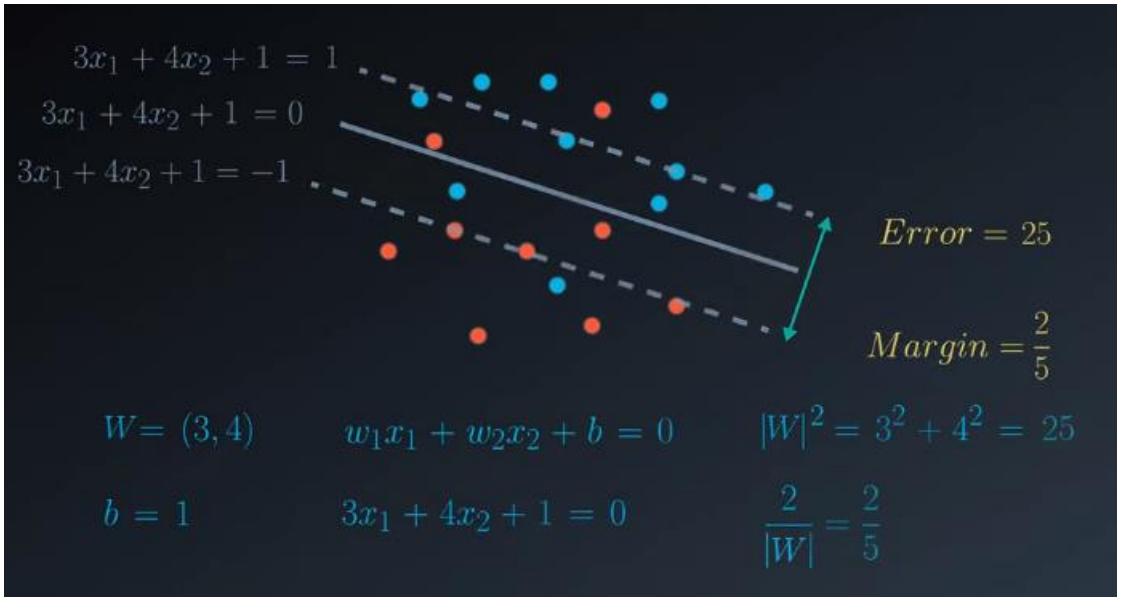




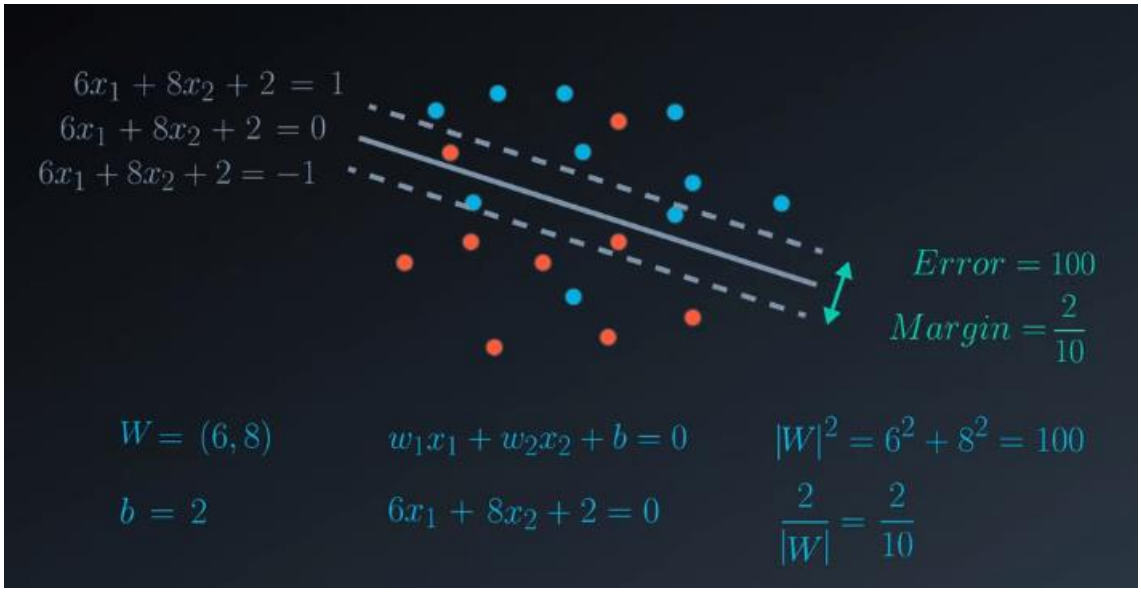
Margin Error



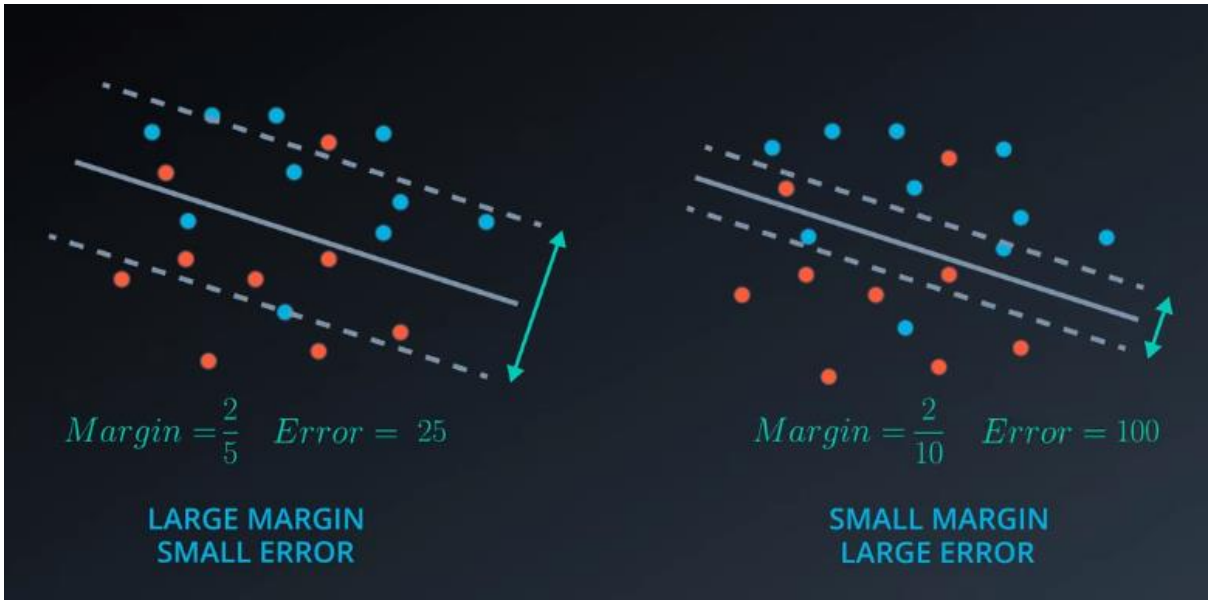
Example I:



Example II:

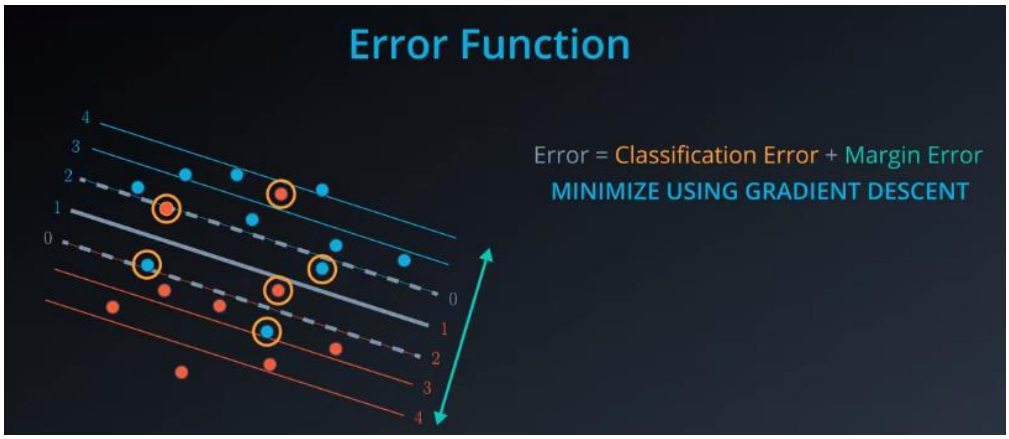


Observations from examples:



This is the exact same error given by the L2 regularisation.

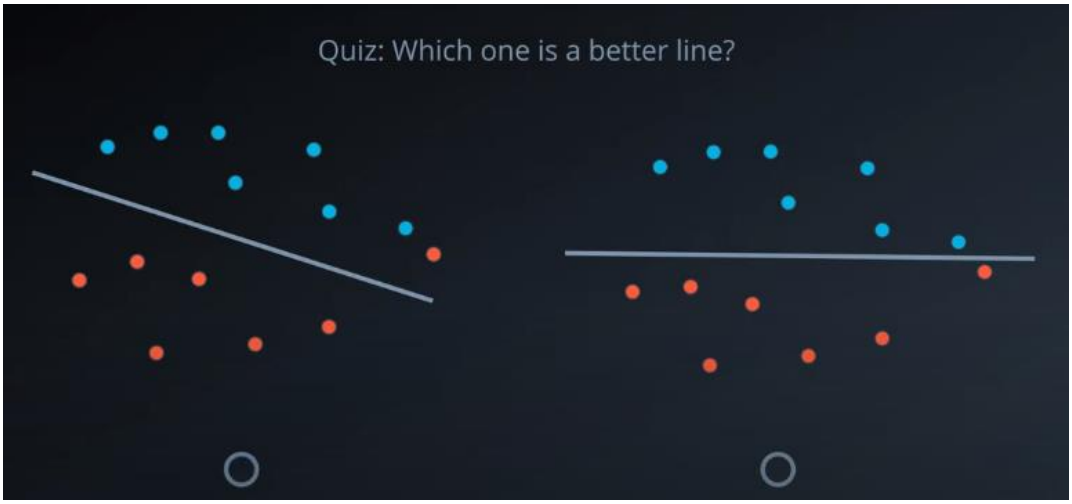
Error Function

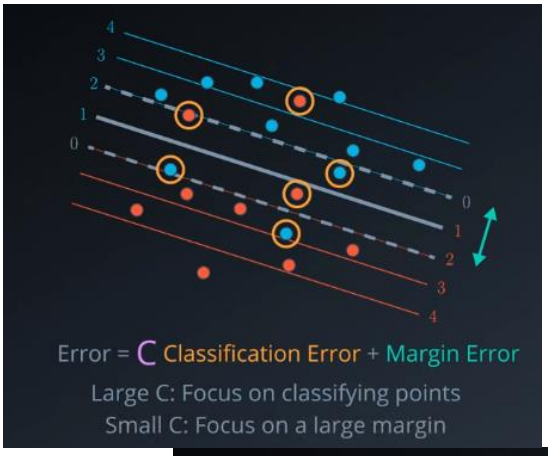


The c Parameter

Depending upon the situation, we can say both the lines have their own pro and cons.

So, There is a need of flexibility here.



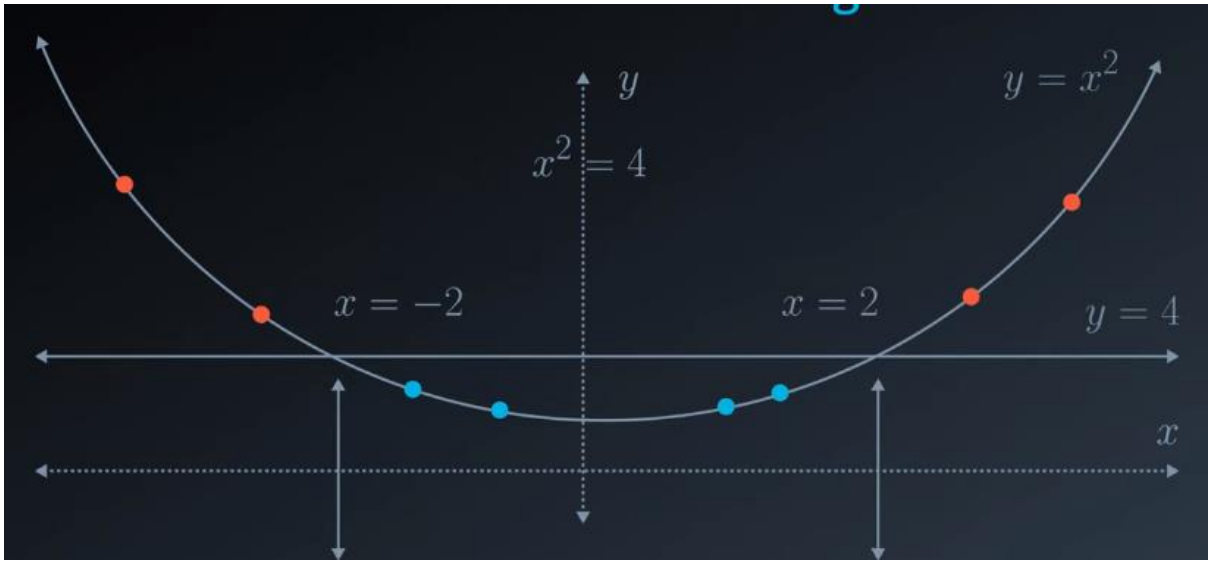


- C is hyperparameter than can be tuned using GridSearch, etc.

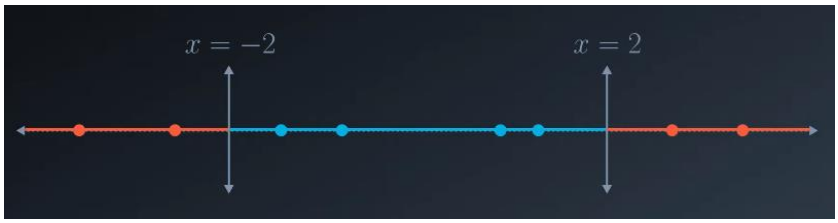
Polynomial Kernal 1

No single line can divide both classes away. So, we can solve this problem by extending it to another dimension.

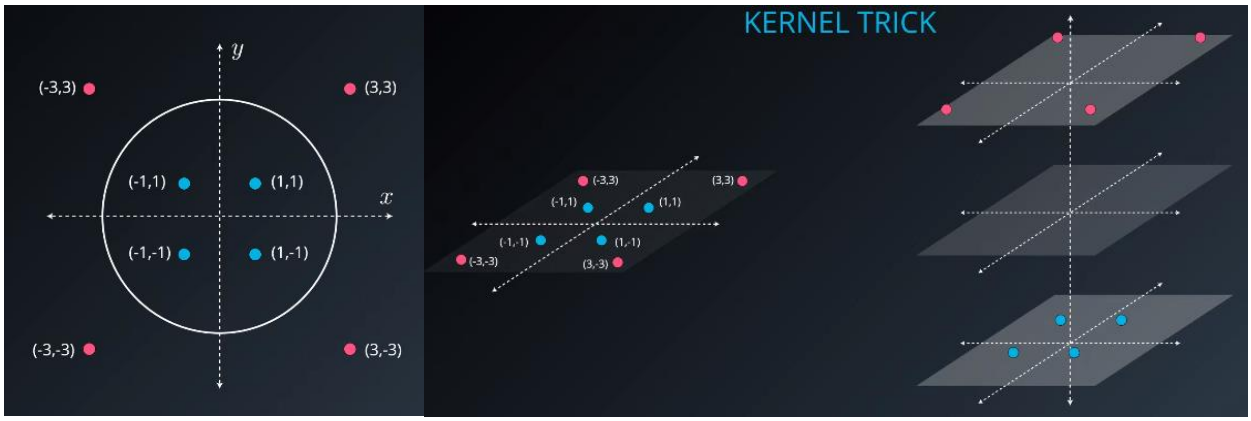
Here:



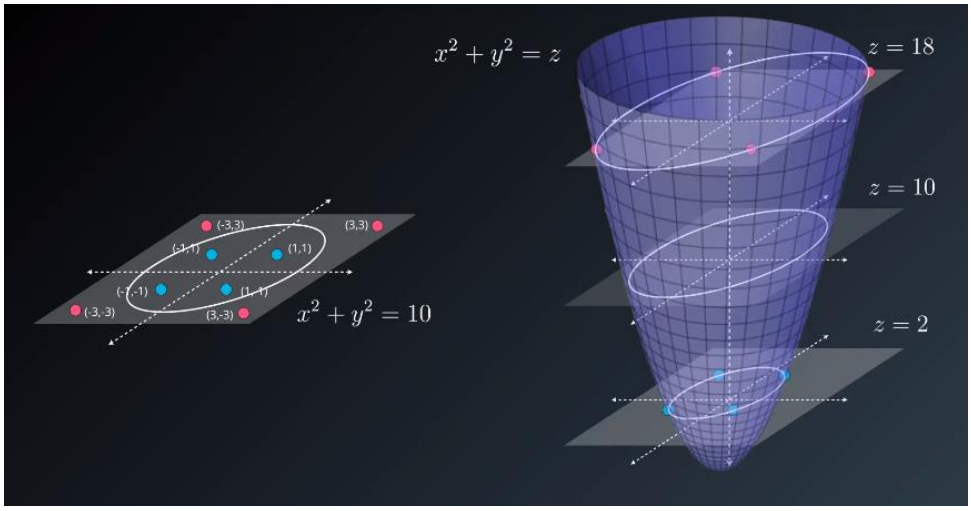
This is called kernel trick, so outcome becomes.



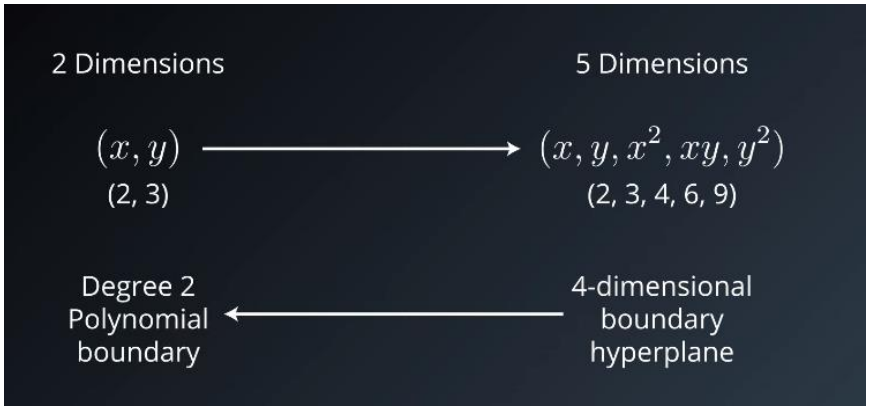
Kernel Trick



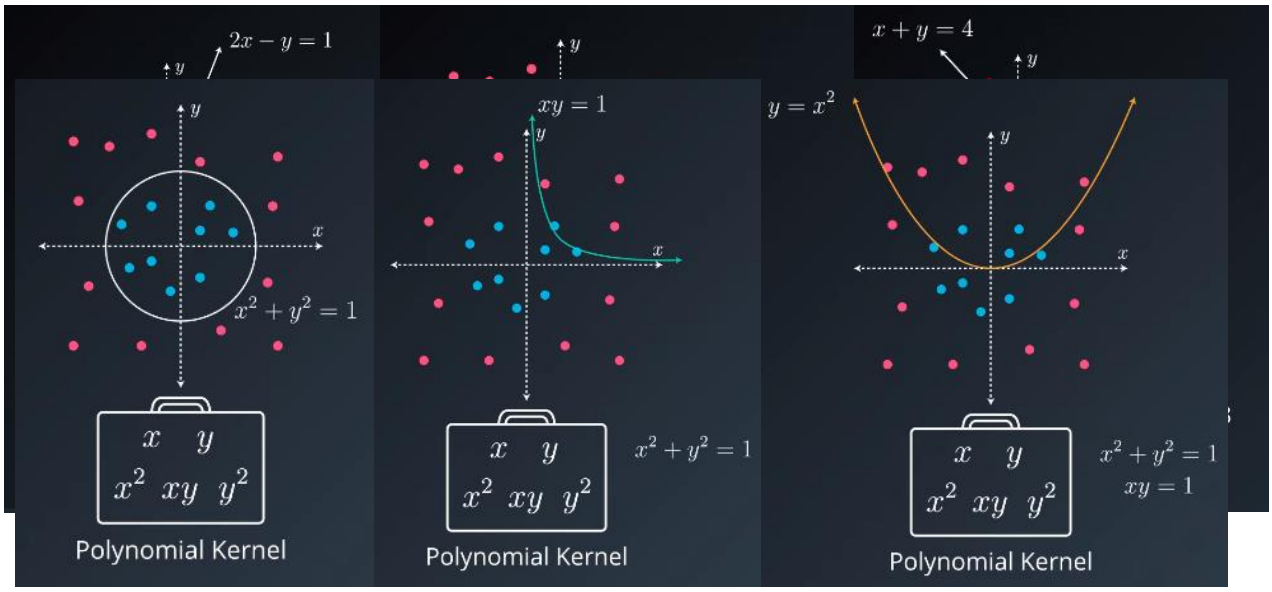
- We have two methods to deal with such situation, one circular where do keep the dimensionality same. Other is transferring the point to another plane, that increases dimensions. In fact they are the same methods.



- This is called polynomial kernel

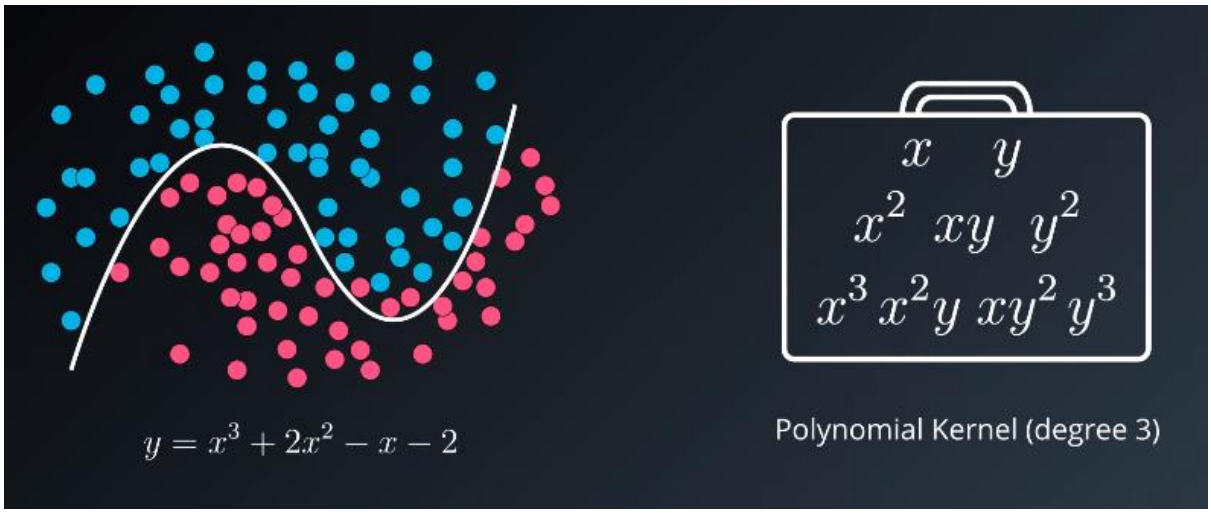


- So, the kernel is set of functions that help us in finding the cut. Above the parabolic 3D structure if the kernel and the circle x2 + y2 = 10 is just a part of it.

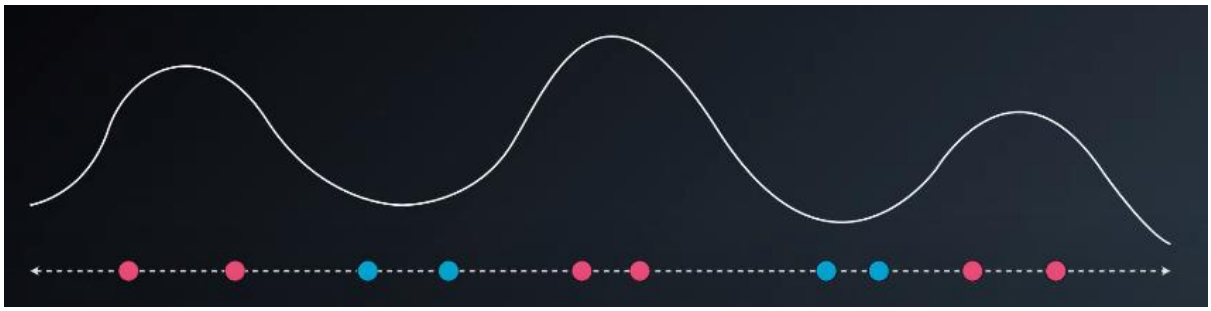


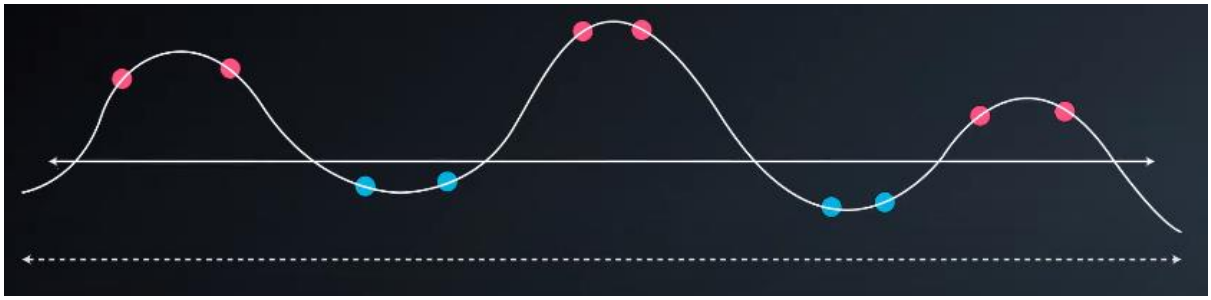
for polynomial kernel we have more options like x, y, x2, y2, & xy so we get following equations:

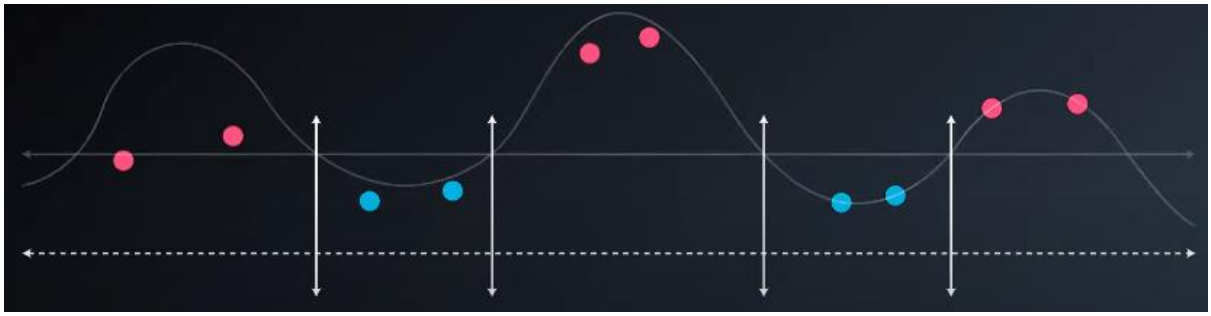
Here is the function with kernel degree 3:

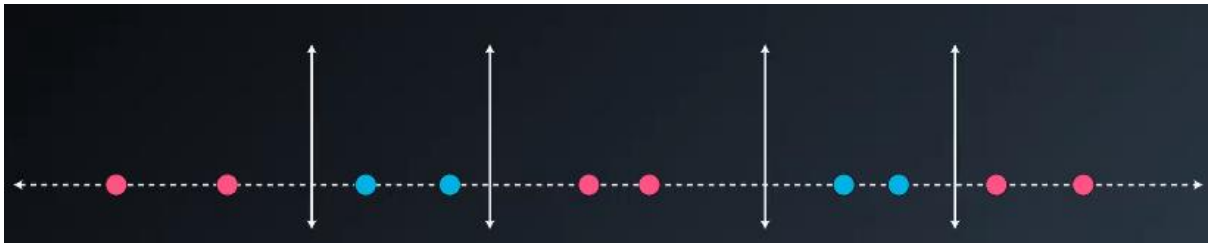


Radial Basic Functions:

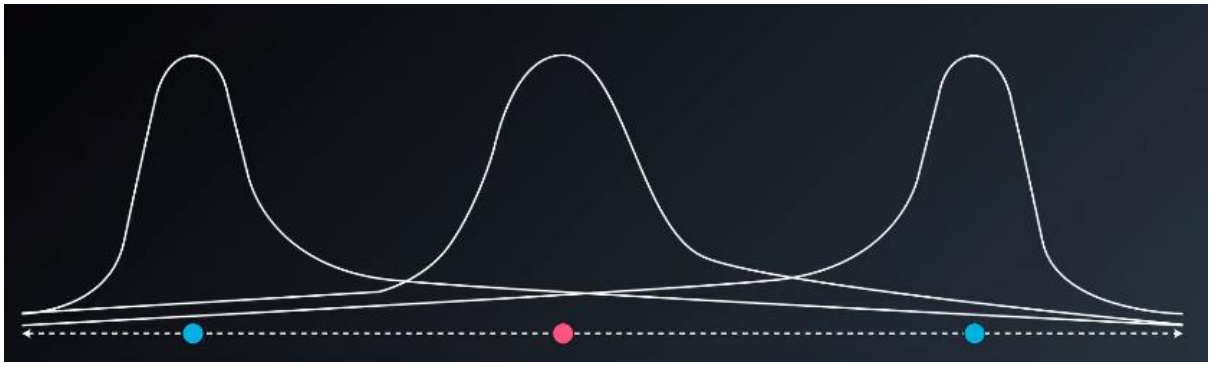


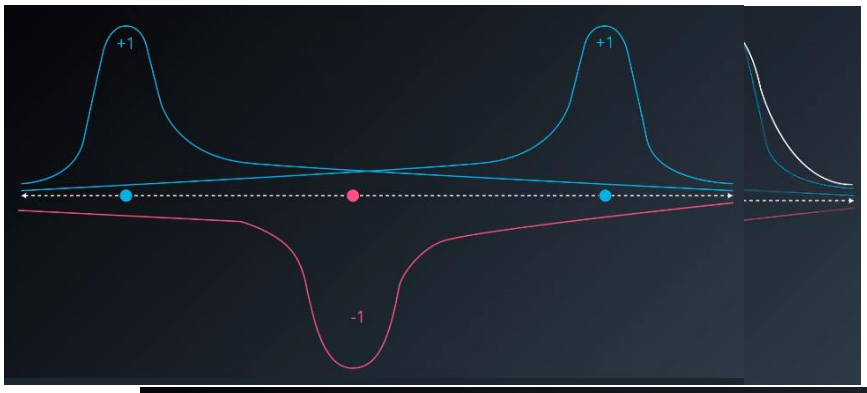


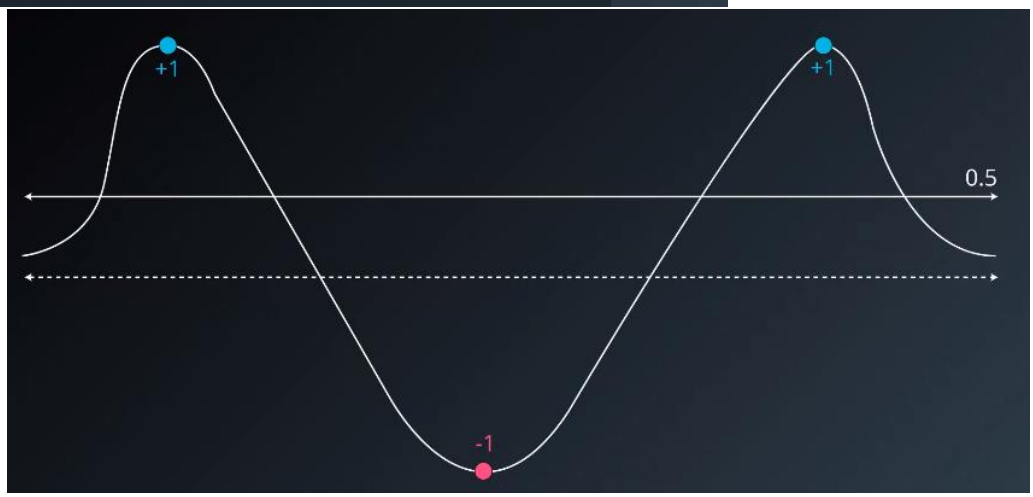


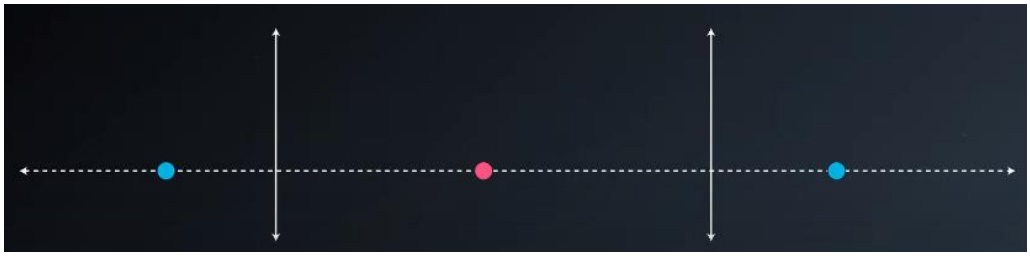


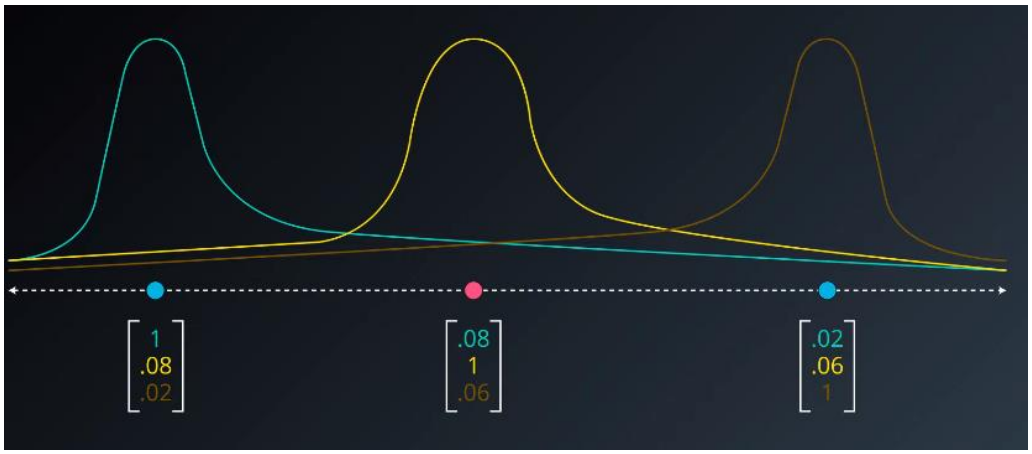
We use RBF kernels to get such mountains:

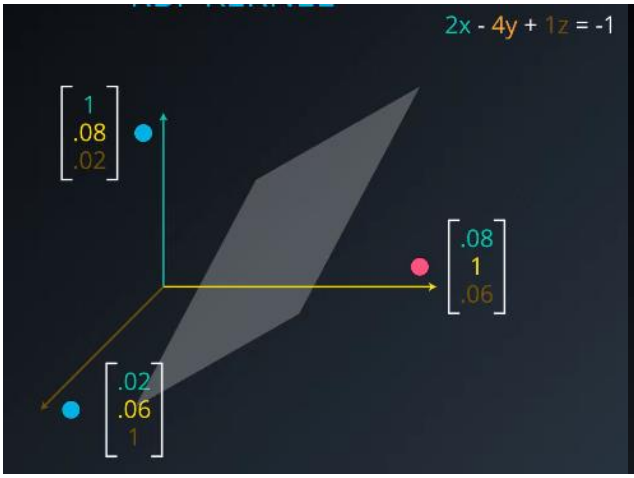


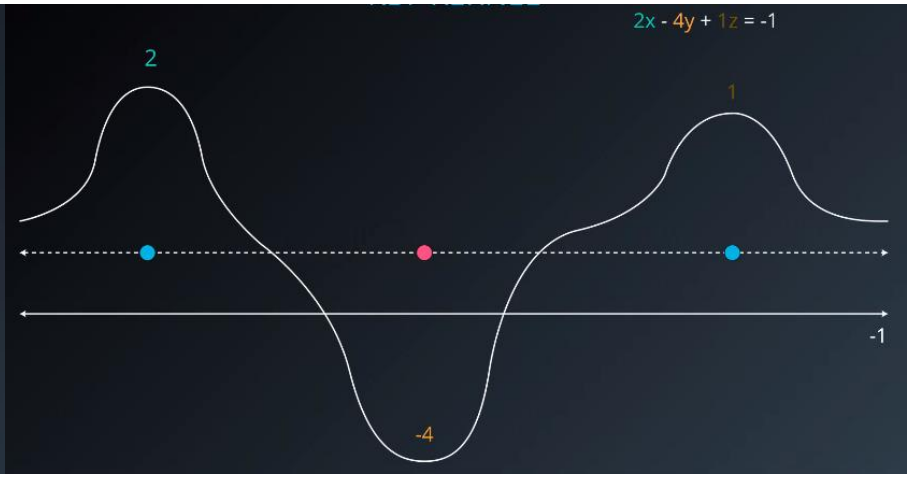


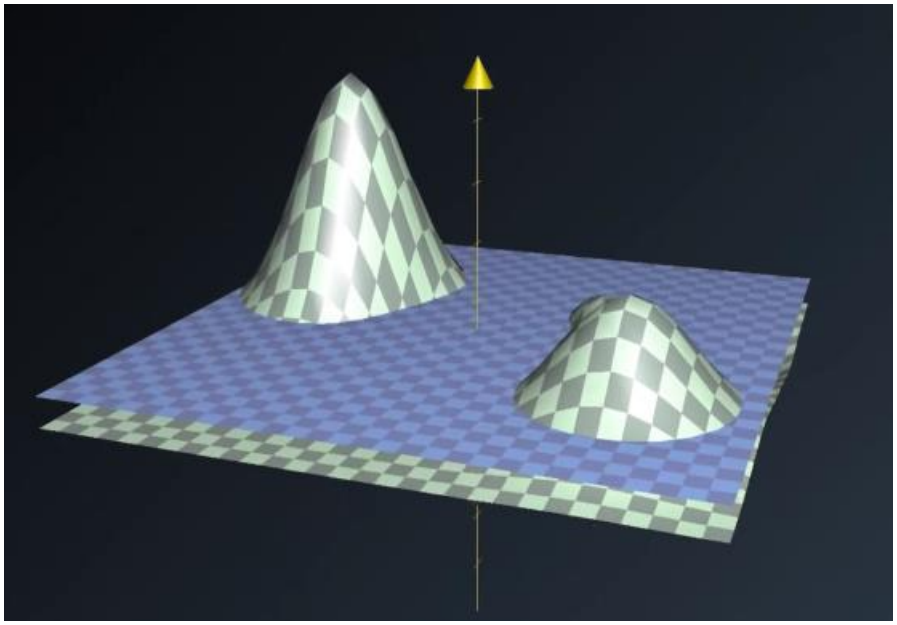





1
.08
.02
.08
1
.06
.02
.06
1

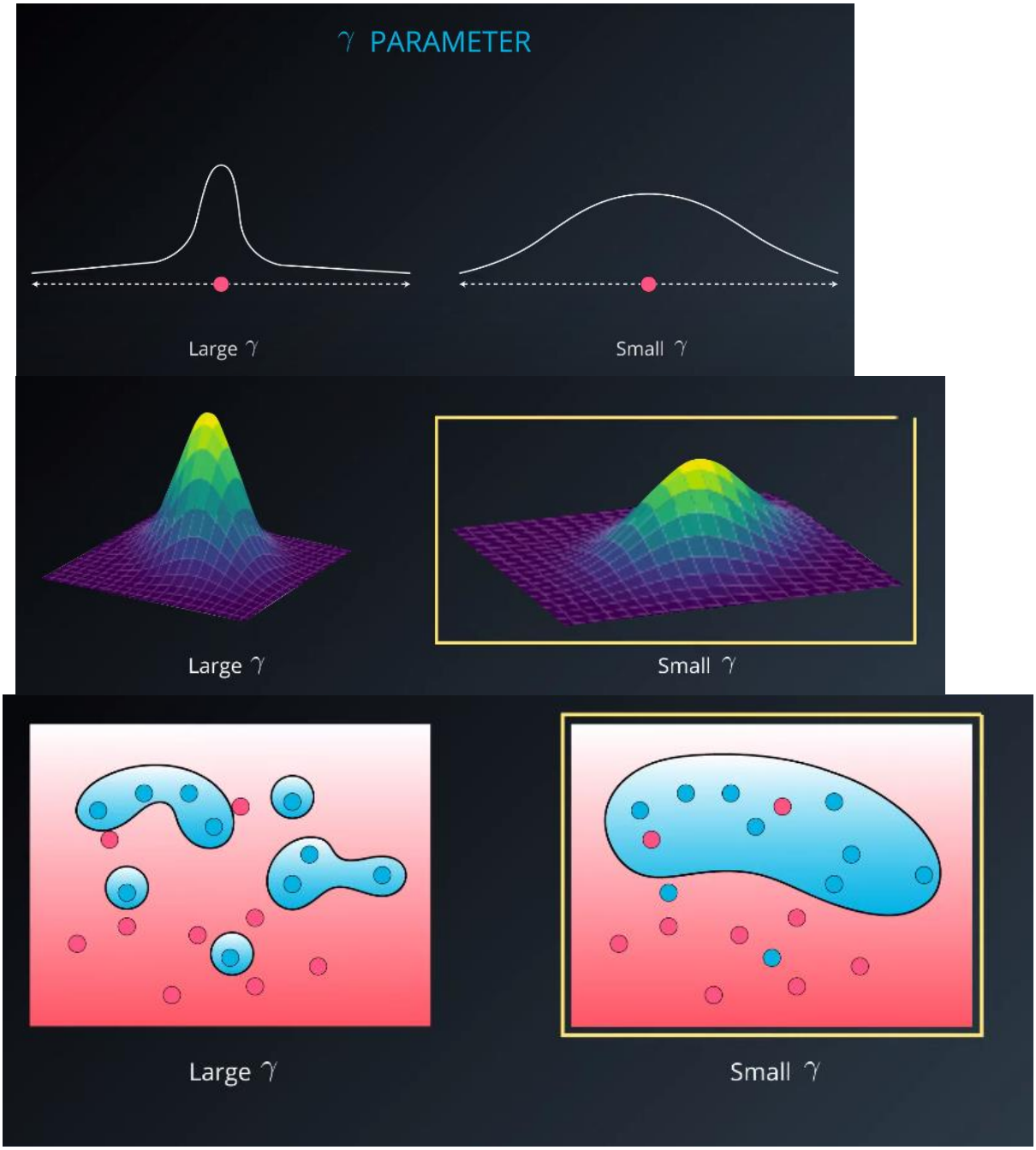

2x - 4y + 1z = -1
1
.08
.02
.08
1
.06
.02
.06
1


2x - 4y + 1z = -1
2
1
-1
-4

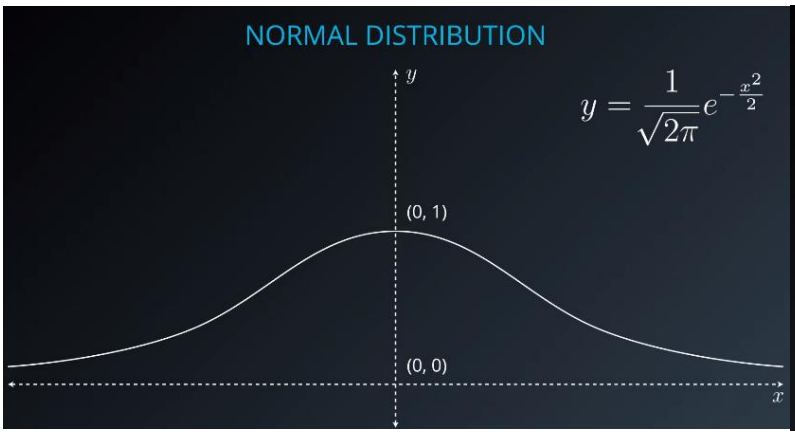


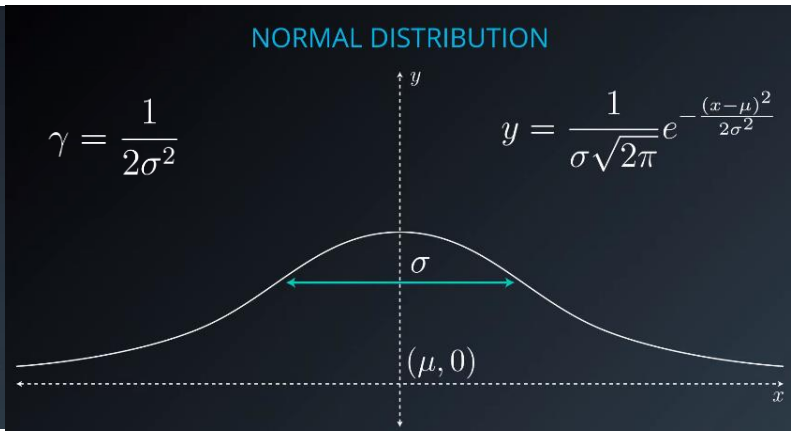
When we project the intersections of the curve on the plane will give us the boundaries that will split our data.

Gamma Parameter



Large gamma causes overfitting, small gamma causes underfitting.





Support Vector Machines in sklearn

Here we'll be using scikit-learn's `SVC` class. This class provides the functions to define and fit the model to your data.

```
>>> from sklearn.svm import SVC
>>> model = SVC()
>>> model.fit(x_values, y_values)
```

Hyperparameters

When we define the model, we can specify the hyperparameters. As we've seen in this section, the most common ones are

- `C`: The C parameter.
- `kernel`: The kernel. The most common ones are 'linear', 'poly', and 'rbf'.
- `degree`: If the kernel is polynomial, this is the maximum degree of the monomials in the kernel.
- `gamma` : If the kernel is rbf, this is the gamma parameter.

Recap:

We saw SVM can be implemented in 3 ways –

- Maximum Margin Classifier
  When your data can be completely separated, the linear version of SVMs attempts to maximize the distance from the linear boundary to the closest points (called the support vectors). For this reason, we saw that in the picture below, the boundary on the left is better than the one on the right.

- Classification with Inseparable Classes
  Unfortunately, data in the real world is rarely completely separable as shown in the above images. For this reason, we introduced a new hyper-parameter called **C**.

The **C** hyper-parameter determines how flexible we are willing to be with the points that fall on the wrong side of our dividing boundary. The value of **C** ranges between 0 and infinity. When **C** is large, you are forcing your boundary to have fewer errors than when it is a small value.

Note: when C is too large for a particular set of data, you might not get convergence at all because your data cannot be separated with the small number of errors allotted with such a large value of C.

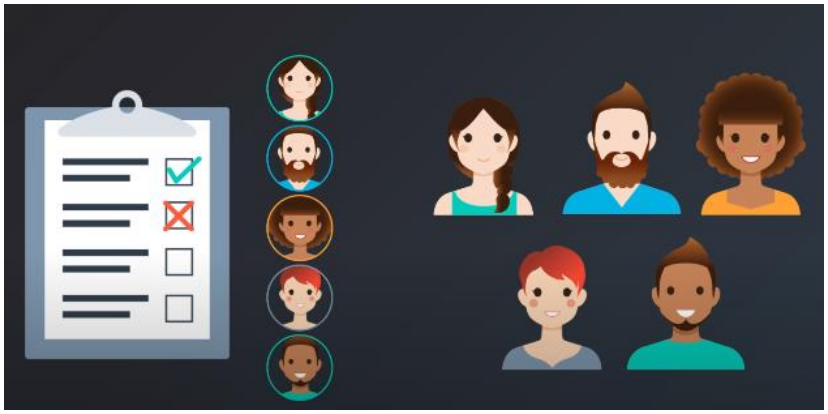
- **Kernels**
  Finally, we looked at what makes SVMs truly powerful, kernels. Kernels in SVMs allow us the ability to separate data when the boundary between them is nonlinear. Specifically, you saw two types of kernels:

  - polynomial
  - rbf

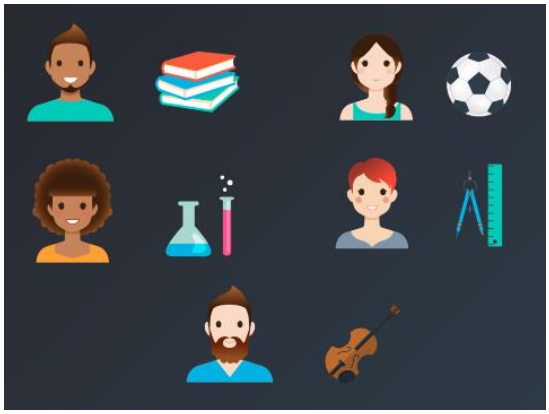
  By far the most popular kernel is the **rbf** kernel (which stands for radial basis function). The rbf kernel allows you the opportunity to classify points that seem hard to separate in any space. This is a density based approach that looks at the closeness of points to one another. This introduces another hyper-parameter **gamma**. When **gamma** is large, the outcome is similar to having a large value of **C**, that is your algorithm will attempt to classify every point correctly. Alternatively, small values of **gamma** will try to cluster in a more general way that will make more mistakes, but may perform better when it sees new data.

Ensemble Methods

Bagging- taking avg of all responses from everyone



Boosting: consider experts in their fields.



Ensembles

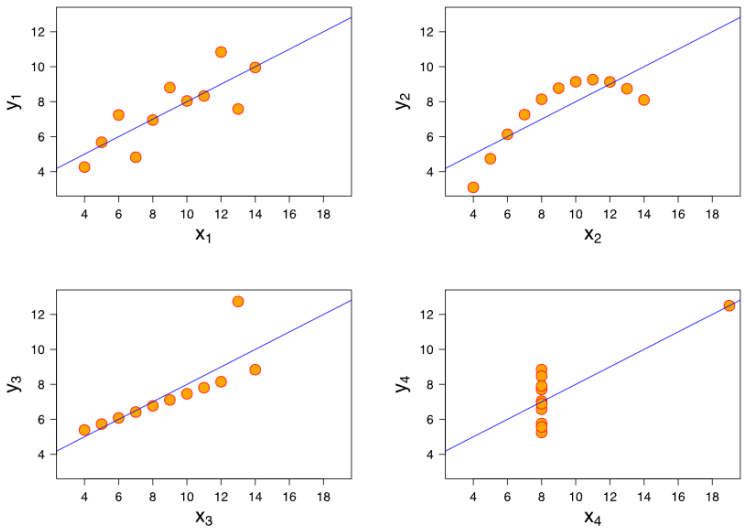
This whole lesson (on ensembles) is about how we can combine (or ensemble) the models you have already seen in a way that makes the combination of these models better at predicting than the individual models.

Commonly the "weak" learners you use are decision trees. In fact the default for most ensemble methods is a decision tree in sklearn. However, you can change this value to any of the models you have seen so far.

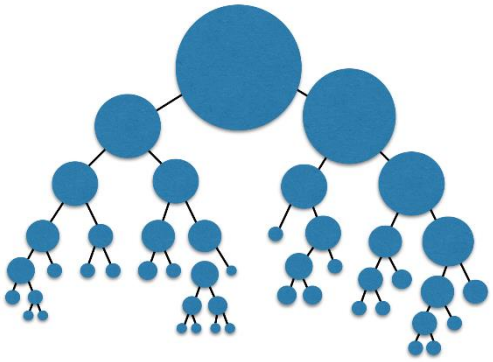
Why ensembling the learners together?

There are two competing variables in finding a well-fitting machine learning model: *Bias* and *Variance*.

Bias: When a model has high bias, this means that means it doesn't do a good job of bending to the data. An example of an algorithm that usually has high bias is linear regression. Even with completely different datasets, we end up with the same line fit to the data. When models have high bias, this is bad.



Variance: When a model has high variance, this means that it changes drastically to meet the needs of every point in our dataset. Linear models like the one above has low variance, but high bias. An example of an algorithm that tends to have high variance and low bias is a decision tree (especially decision trees with no early stopping parameters). A decision tree, as a high variance algorithm, will attempt to split every point into its own branch if possible. This is a trait of high variance, low bias algorithms - they are extremely flexible to fit exactly whatever data they see.



Introducing randomness in Ensembles

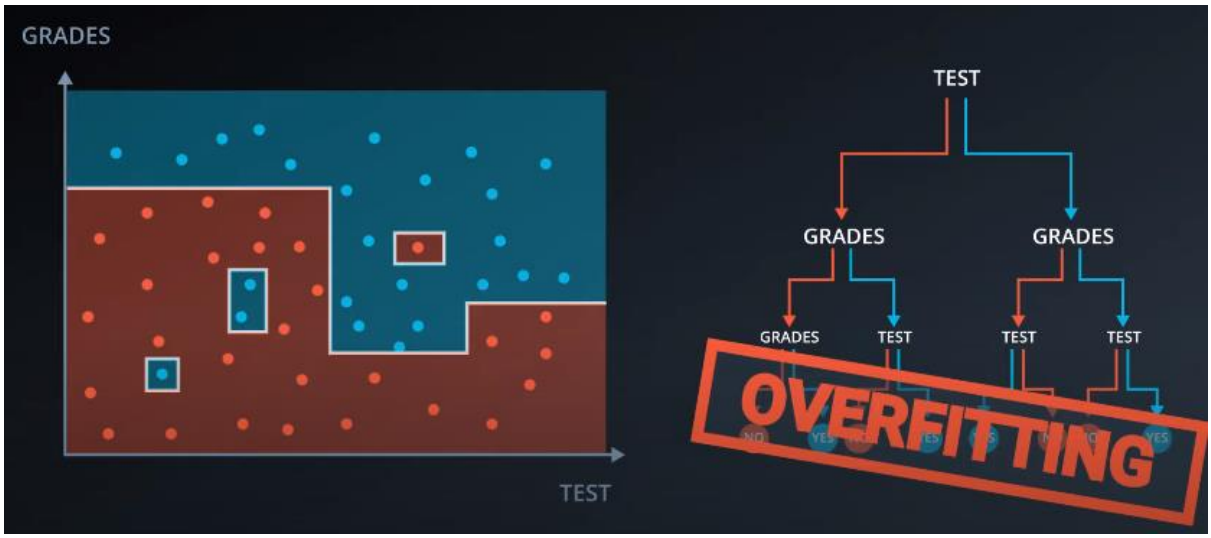
Another method that is used to improve ensemble methods is to introduce randomness into high variance algorithms before they are ensembled together. The introduction of randomness combats the tendency of these algorithms to overfit (or fit directly to the data available). There are two main ways that randomness is introduced:

1. **Bootstrap the data** - that is, sampling the data with replacement and fitting your algorithm to the sampled data.
2. **Subset the features** - in each split of a decision tree or with each algorithm used in an ensemble, only a subset of the total possible features are used.

In fact, these are the two random components used in the next algorithm you are going to see called **random forests**.

Random Forest

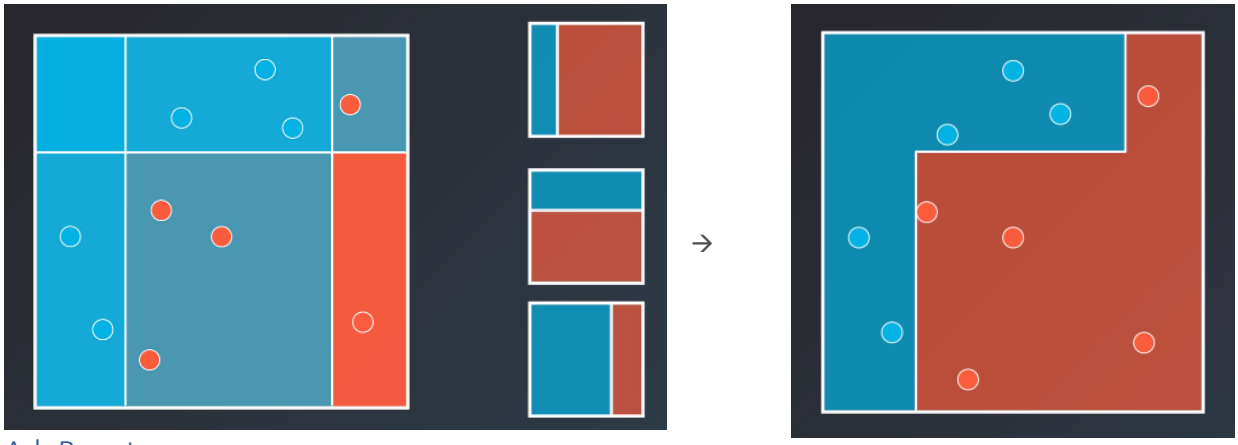
Decision tree tend to overfit a lot, because it seems they memorise the branching.



Solution? Picking up random columns and work with them.

Bagging

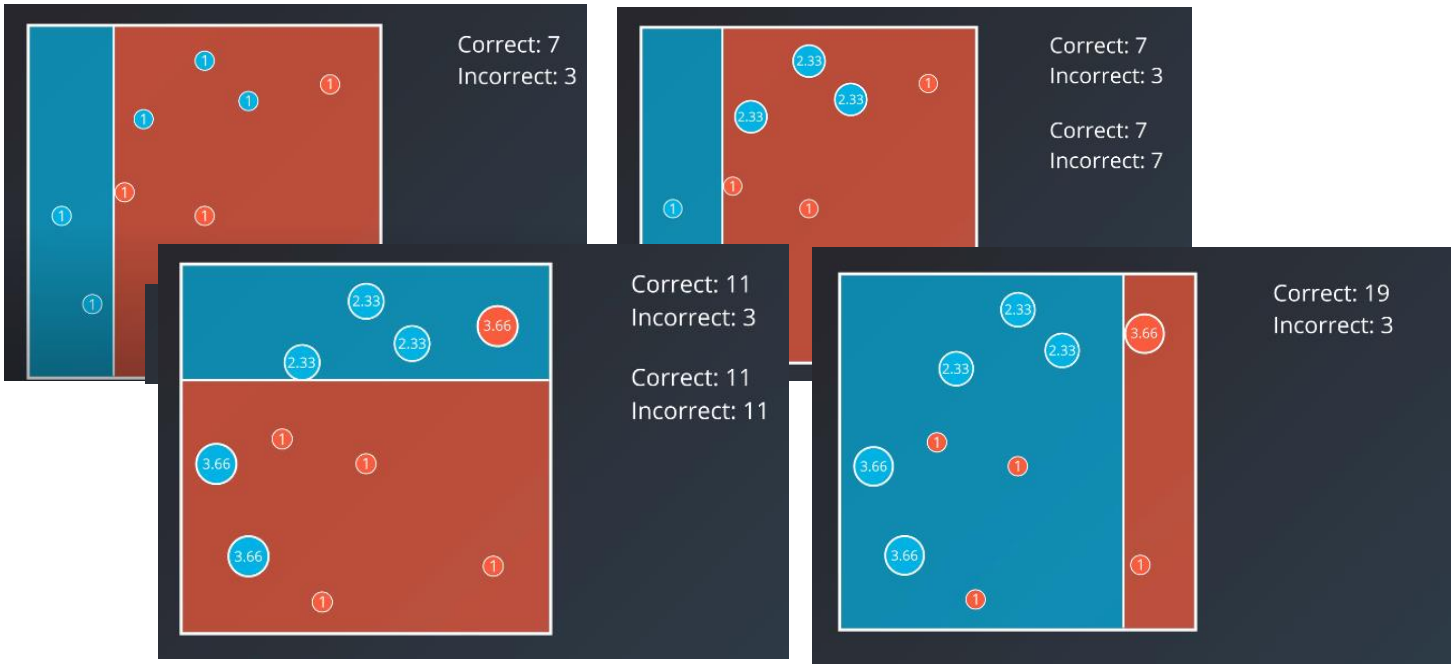
We randomly pick any set of data, make partitions based on each set and then choose the overlap and give the region to most voted out of them.



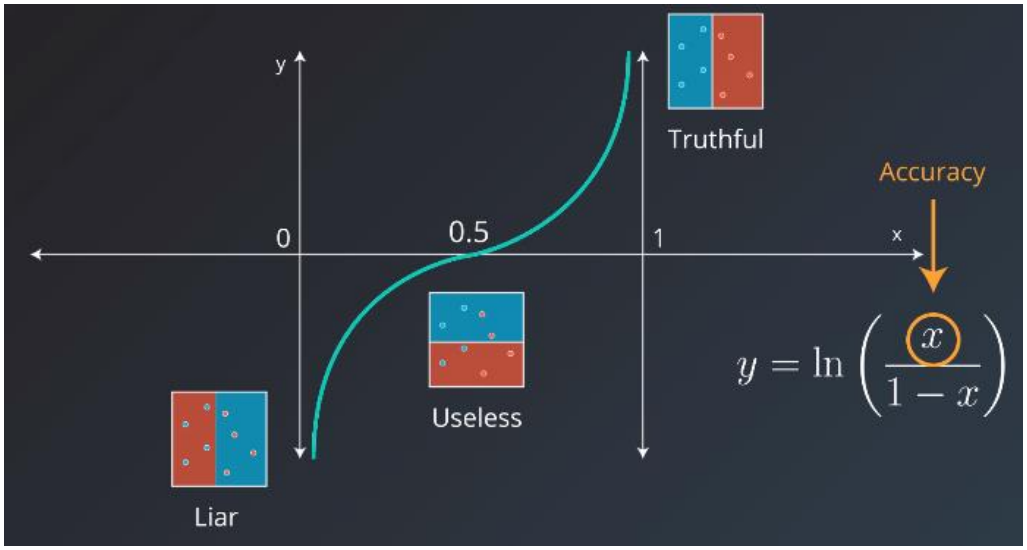
AdaBoost

Here we choose the first partition to minimise the number of errors, then punish the mistakes by enlarging the errors, it has made. In the next round it tries to mend the mistakes made in 1st step. In 3rd step it tries to sort out mistakes from the 2nd step.

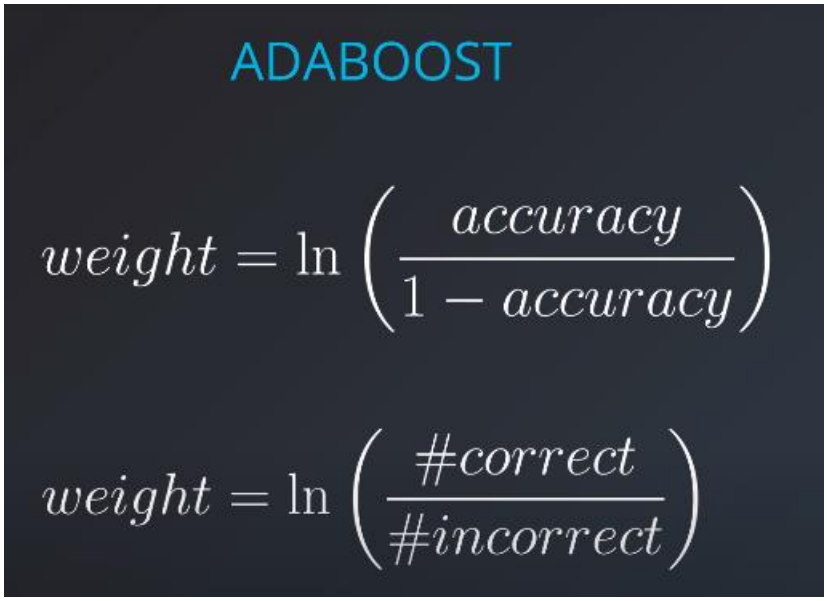
<u>Weighting the data</u>



Weighing the Models

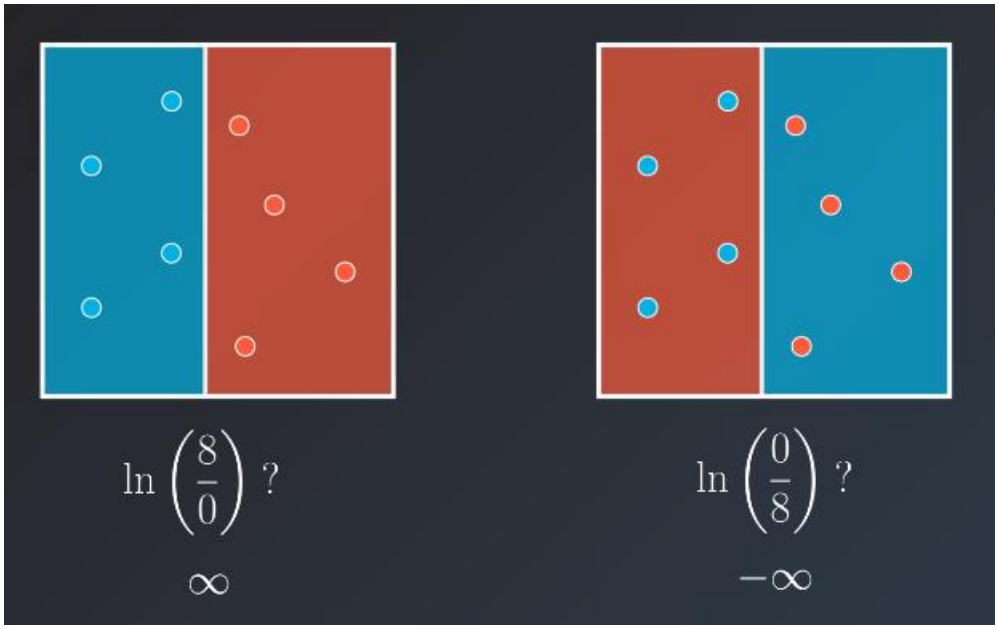


$$weight = \ln\left(\frac{accuracy}{1 - accuracy}\right)$$

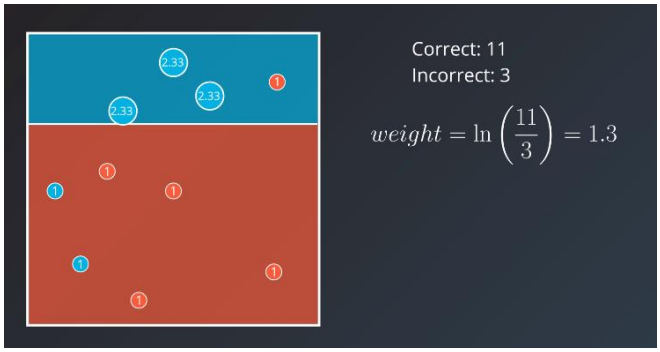


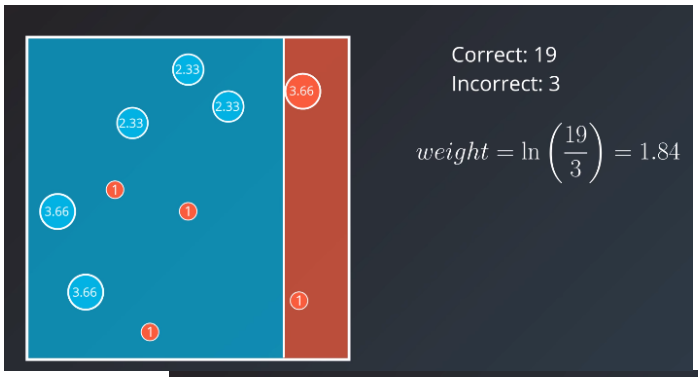
Additional Resources on Boosting

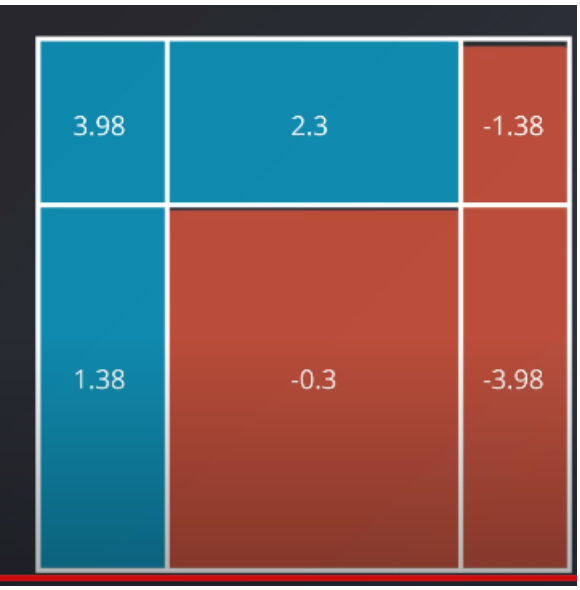
1. **The original paper** - A link to the original paper on boosting by Yoav Freund and Robert E. Schapire.
2. **An explanation about why boosting is so important** - A great article on boosting by a Kaggle master, Ben Gorman.
3. **A useful Quora post** - A number of useful explanations about boosting.



Combining the models







FINAL output for the model. This is how Adaboost works.

AdaBoost in sklearn

Hyperparameters

When we define the model, we can specify the hyperparameters. In practice, the most common ones are

- `base_estimator`: The model utilized for the weak learners (**Warning:** Don't forget to import the model that you decide to use for the weak learner).
- `n_estimators`: The maximum number of weak learners used.

Quick recap:

- **High Bias, Low Variance** models tend to underfit data, as they are not flexible. **Linear models** fall into this category of models.
- **High Variance, Low Bias** models tend to overfit data, as they are too flexible. **Decision trees** fall into this category of models.

There were two randomization techniques you saw to combat overfitting:

1. **Bootstrap the data** - that is, sampling the data with replacement and fitting your algorithm and fitting your algorithm to the sampled data.
2. **Subset the features** - in each split of a decision tree or with each algorithm used an ensemble only a subset of the total possible features are used.

Techniques

You saw a number of ensemble methods in this lesson including:

- **BaggingClassifier**
- **RandomForestClassifier**
- **AdaBoostClassifier**

Another really useful guide for ensemble methods can be found **in the documentation here**. These methods can also all be extended to regression problems, not just classification.

Additional Resources

Additionally, here are some great resources on AdaBoost if you'd like to learn some more!

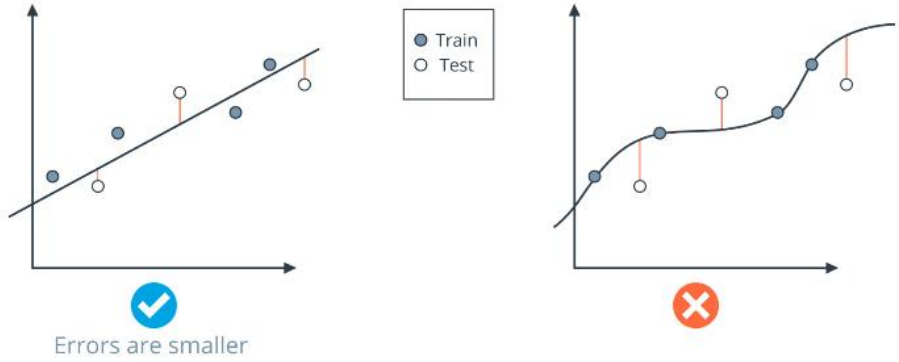
- Here is the original **paper** from Freund and Schapire.
- A follow-up **paper** from the same authors regarding several experiments with Adaboost.
- A great **tutorial** by Schapire.

Model Evaluation

- How well the model is doing?
- How do we improve the model based on the metrics?

Regression and Classification

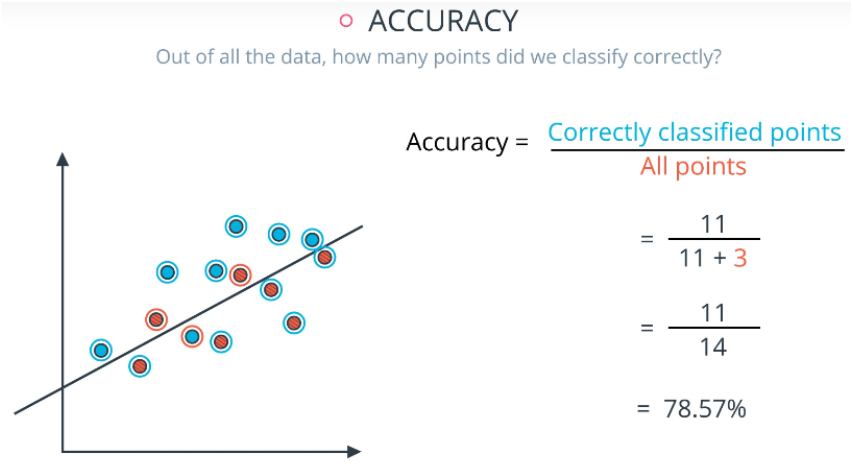
TESTING



Confusion Matrix

TP,TN,FP,FN

Accuracy



Precision and Recall

Precision in different cases:

DIAGNOSIS

PATIENTS	Diagnosed Sick	Diagnosed Healthy
Sick	1000	200 ⊗
Healthy	800	9000

PRECISION: OUT OF THE PATIENTS WE DIAGNOSED WITH AN ILLNESS, HOW MANY DID WE CLASSIFY CORRECTLY?

$$\text{PRECISION} = \frac{1{,}000}{1{,}000 + 800} = 55.6\%$$

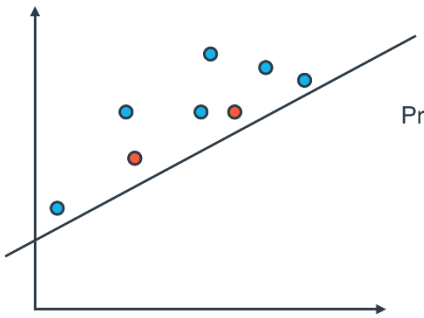
FOLDER

EMAIL	Sent to Spam Folder	Sent to Inbox
Spam	100	170
Not Spam	30 ⊗	700

OUT OF ALL THE E-MAILS SENT TO THE SPAM FOLDER, HOW MANY WERE ACTUALLY SPAM?

$$\text{PRECISION} = \frac{100}{100 + 30} = 76.9\%$$

Precision



Precision: Out of the points we've predicted to be positive, how many are correct?

$$\text{Precision} = \frac{\text{True positives}}{\text{True positives} + \text{False Positives}} = \frac{6}{6 + 2} = \frac{6}{8} = 75\%$$

Recall

DIAGNOSIS

PATIENTS	Diagnosed Sick	Diagnosed Healthy
Sick	1000	200 ⊗
Healthy	800	8000

OUT OF THE SICK PATIENTS, HOW MANY DID WE CORRECTLY DIAGNOSE AS SICK?

$$\text{RECALL} = \frac{1{,}000}{1{,}000 + 200} = 83.3\%$$

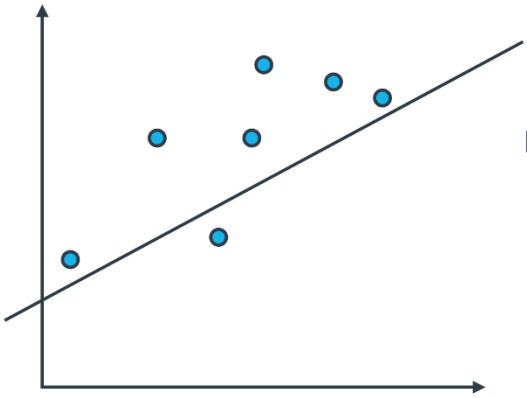
FOLDER

EMAIL	Sent to Spam Folder	Sent to Inbox
Spam	100	170
Not Spam	30 ⊗	700

OUT OF ALL THE SPAM E-MAILS, HOW MANY WERE CORRECTLY SENT TO THE SPAM FOLDER?

$$\text{Recall} = \frac{100}{100 + 170} = 37\%$$

Recall

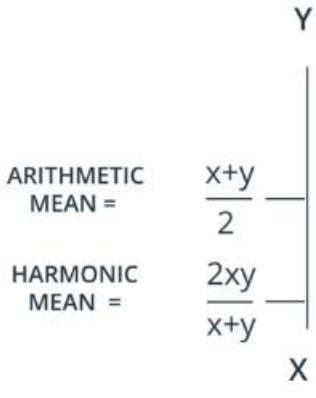


Recall: Out of the points labelled positive, how many did we correctly predict?

$$\text{Recall} = \frac{\text{True positives}}{\text{True positives} + \text{False Negatives}} = \frac{6}{6 + 1} = \frac{6}{7} = 85.7\%$$

F1 score

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$



PRECISION = 1
RECALL = 0
AVERAGE = 0.5
HARMONIC MEAN = 0

PRECISION = 0.2
RECALL = 0.8
AVERAGE = 0.5
HARMONIC MEAN = 0.32

~~ARITHMETIC MEAN (PRECISION, RECALL)~~

F_1 SCORE = HARMONIC MEAN (PRECISION, RECALL)

Considering X<Y, HM lies near to the smallest value. It is always less than AM.

F-beta score

Suppose we want our model to care more about Precision than Recall, or other way around. So, Fbeta in introduced.

$$F_1 \text{ SCORE} = 2 \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$F_\beta \text{ SCORE} = (1+\beta^2) \frac{\text{Precision} * \text{Recall}}{\beta^2 * \text{Precision} + \text{Recall}}$$

PRECISION $F_{0.5}$ SCORE F_1 SCORE F_2 SCORE RECALL

MODEL	F-BETA SCORE
Spaceship	2
Notifications	1
Promotional Material	0.5

Example:

Out of the following three models, which one should have an F-beta score of 2, 1, and 0.5? Match each model with its corresponding score.

- Detecting malfunctioning parts in a spaceship
- Sending phone notifications about videos a user may like
- Sending promotional material in the mail to potential clients

Solution:

- For the spaceship model, we can't really afford any malfunctioning parts, and it's ok if we overcheck some of the parts that are working well. Therefore, this is a **high recall** model, so we associate it with beta = 2.
- For the notifications model, since it's free to send them, we won't get harmed too much if we send them to more people than we need to. But we also shouldn't overdo it, since it will annoy the users. We also would like to find as many interested users as we can. Thus, this is

a model which should have a decent **precision** and a decent **recall**. Beta = 1 should work here.

- For the Promotional Material model, since it costs us to send the material, we really don't want to send it to many people that won't be interested. Thus, this is a high **precision** model. Thus, beta = 0.5 will work here.

Boundaries in the F-beta score

Note that in the formula for F_β score, if we set $\beta = 0$, we get

$F_0 = (1+0^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{0 \cdot \text{Precision} + \text{Recall}} = \frac{\text{Precision} \cdot \text{Recall}}{\text{Recall}} = \text{Precision}$. Therefore, the minimum value of β is zero, and at this value, we get the precision.

Now, notice that if N is really large, then

$$F_\beta = (1+N^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{N^2 \cdot \text{Precision} + \text{Recall}} = \frac{\text{Precision} \cdot \text{Recall}}{\frac{N^2}{1+N^2}\text{Precision} + \frac{1}{1+N^2}\text{Recall}}.$$

As N goes to infinity, we can see that $\frac{1}{1+N^2}$ goes to zero, and $\frac{N^2}{1+N^2}$ goes to 1.

Therefore, if we take the limit, we have

$$\lim_{N \to \infty} F_N = \frac{\text{Precision} \cdot \text{Recall}}{1 \cdot \text{Precision} + 0 \cdot \text{Recall}} = \text{Recall}.$$

Thus, to conclude, the boundaries of beta are between 0 and \infty.∞.

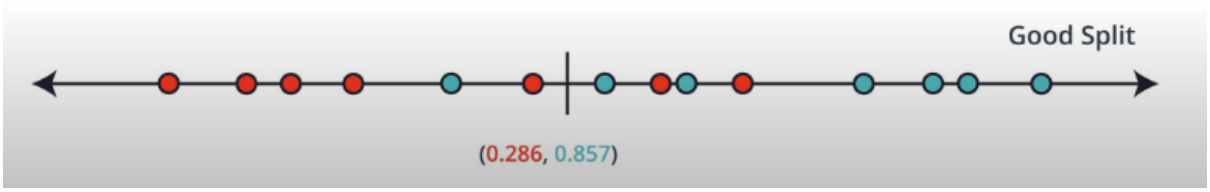
- If \beta = 0,β=0, then we get **precision**.
- If \beta = \infty,β=∞, then we get **recall**.
- For other values of \beta,β, if they are close to 0, we get something close to precision, if they are large numbers, then we get something close to recall, and if \beta = 1,β=1, then we get the **harmonic mean** of precision and recall.

ROC curve

Lets consider one cut and calculate TP and FP wrt that point.

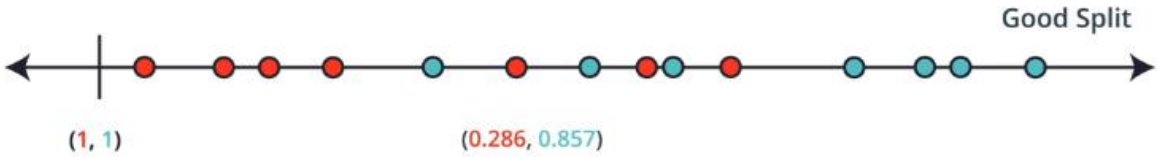
$$\text{True Positive Rate} = \frac{\text{TRUE POSITIVES}}{\text{ALL POSITIVES}} = \frac{6}{7} =$$

$$\text{False Positive Rate} = \frac{\text{FALSE POSITIVES}}{\text{ALL NEGATIVES}} = \frac{2}{7} =$$



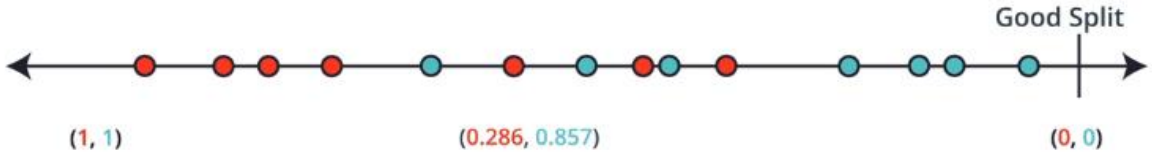
$$\text{True Positive Rate} = \frac{\text{TRUE POSITIVES}}{\text{ALL POSITIVES}} = \frac{7}{7} =$$

$$\text{False Positive Rate} = \frac{\text{FALSE POSITIVES}}{\text{ALL NEGATIVES}} = \frac{7}{7} =$$

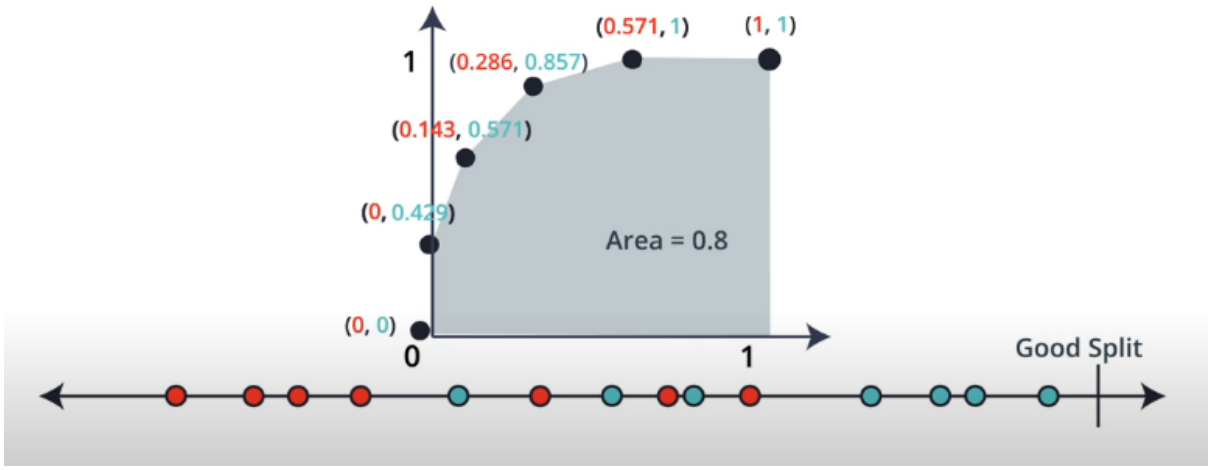


$$\text{True Positive Rate} = \frac{\text{TRUE POSITIVES}}{\text{ALL POSITIVES}} = \frac{0}{7} =$$

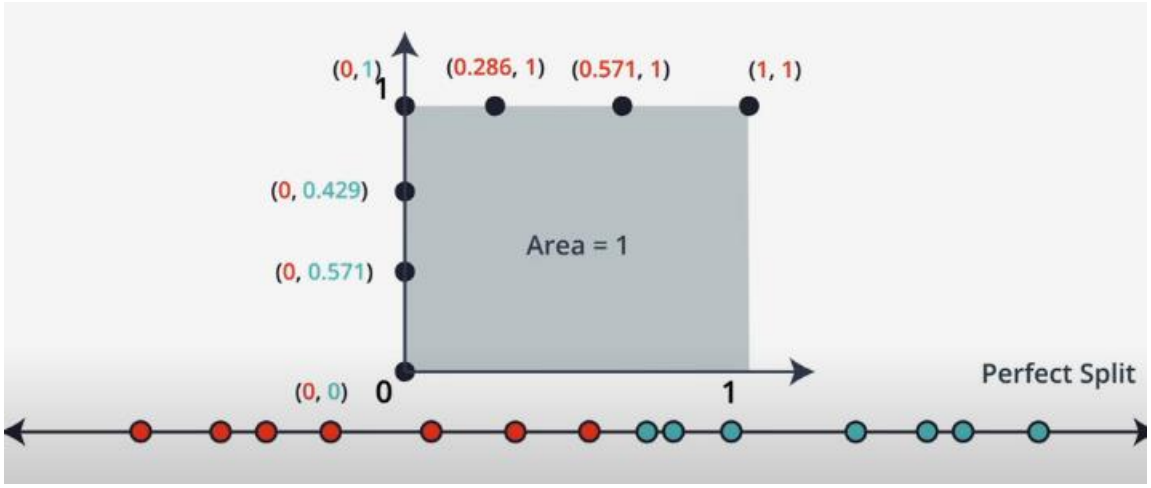
$$\text{False Positive Rate} = \frac{\text{FALSE POSITIVES}}{\text{ALL NEGATIVES}} = \frac{0}{7} =$$



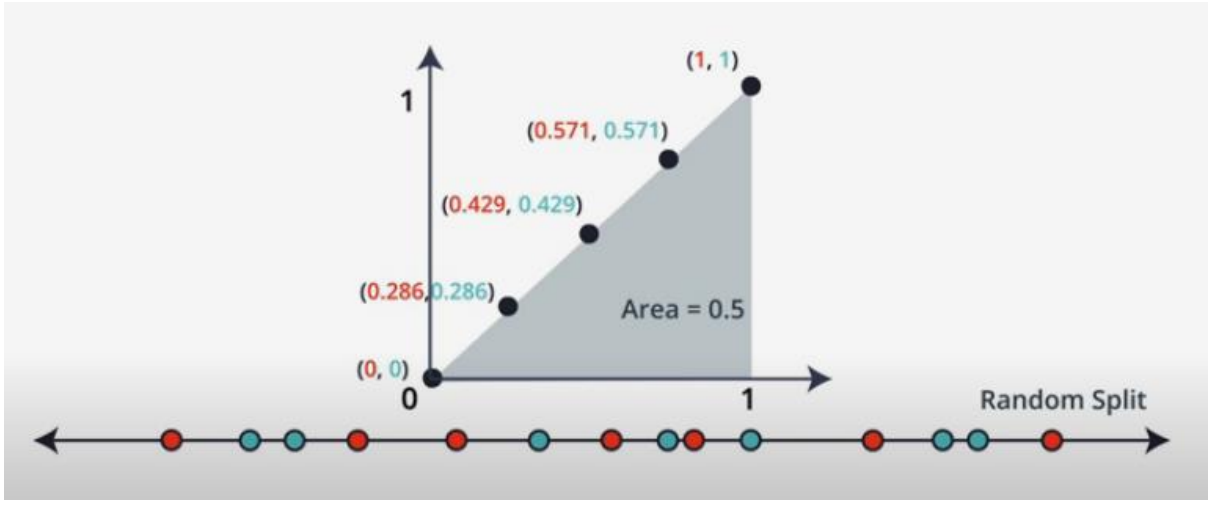
So we calculate rates at multiple cuts in line.



Considering the perfect split we get:



For random split:



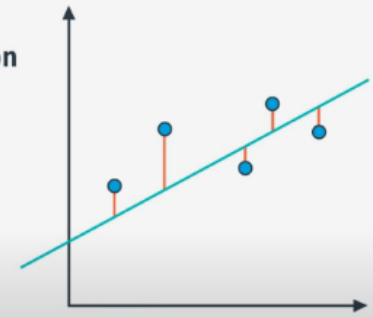
CAN the area under the curve < 0.5?

YES! When we have more blue points in red area and vice versa.

Regression Metrics

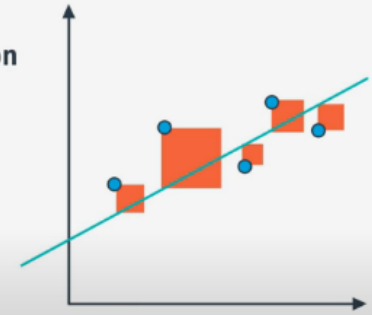
- Mean absolute Error

```
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import LinearRegression

classifier = LinearRegression()
classifier.fit(X,y)

guesses = classifier.predict(X)

error = mean_absolute_error(y, guesses)
```

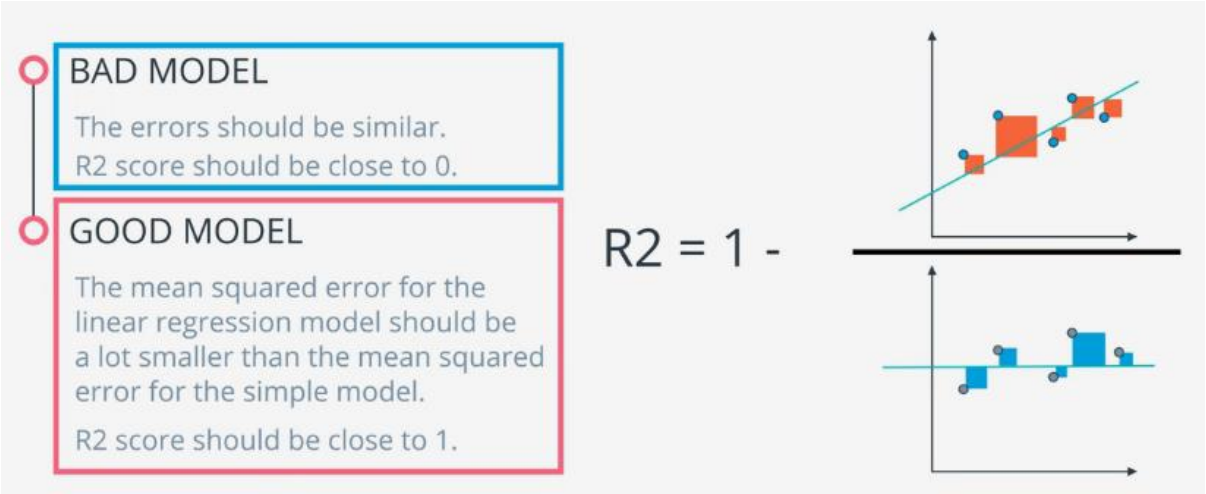


    - The problem with MEA is; the absolute value function not differentiable; hence GD cant be used.
- Mean squared error

```
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression

classifier = LinearRegression()
classifier.fit(X,y)

guesses = classifier.predict(X)

error = mean_squared_error(y, guesses)
```



- R2 Score
    - Comparing our model to simplest possible model.



RECAP

Classification Measures

If you are fitting your model to predict categorical data (spam not spam), there are different measures to understand how well your model is performing than if you are predicting numeric values (the price of a home).

As we look at classification metrics, note that the **wikipedia page** on this topic is wonderful, but also a bit daunting. I frequently use it to remember which metric does what.
Specifically, you saw how to calculate:

- **Accuracy** - Accuracy is often used to compare models, as it tells us the proportion of observations we correctly labeled.

		Actual	
		Spam (Positive)	Not Spam (Negative)
Predicted	Spam (Positive)	True Positive (TP)	False Positive (FP)
	Not Spam (Negative)	False Negative (FN)	True Negative (TN)

$$\text{Accuracy} = \frac{\text{True Positives + True Negatives}}{\text{TP + FP + FN + TN}}$$

  - Often accuracy is not the only metric you should be optimizing on. This is especially the case when you have class imbalance in your data. Optimizing on only accuracy can be misleading in how well your model is truly performing. With that in mind, you saw some additional metrics.

- **Precision** - Precision focuses on the **predicted** "positive" values in your dataset. By optimizing based on precision values, you are determining if you are doing a good job of predicting the positive values, as compared to predicting negative values as positive.

		Actual	
		Spam (Positive)	Not Spam (Negative)
Predicted	Spam (Positive)	True Positive (TP)	False Positive (FP)
	Not Spam (Negative)	False Negative (FN)	True Negative (TN)

$$\text{Precision} = \frac{\text{True Positives}}{\text{TP + FP}}$$

- **Recall** - Recall focuses on the **actual** "positive" values in your dataset. By optimizing based on recall values, you are determining if you are doing a good job of predicting the positive values **without** regard of how you are doing on the **actual** negative values. If you want to perform something similar to recall on the **actual** 'negative' values, this is called `specificity` (TN / (TN + FP)).

Predicted \ Actual	Spam (Positive)	Not Spam (Negative)
Spam (Positive)	True Positive (TP)	False Positive (FP)
Not Spam (Negative)	False Negative (FN)	True Negative (TN)

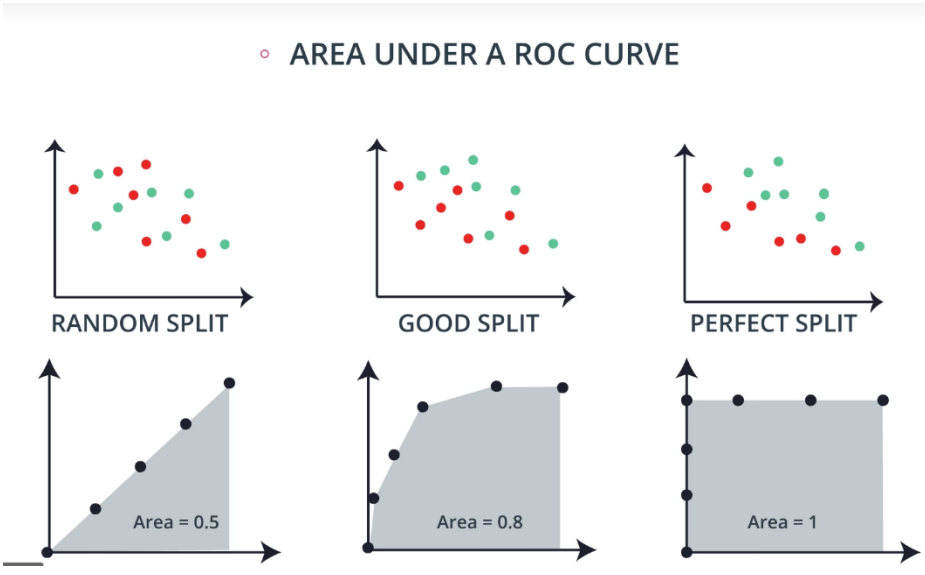
$$\text{Recall} = \frac{\text{True Positives}}{TP + FN}$$

- **F-Beta Score** - In order to look at a combination of metrics at the same time, there are some common techniques like the F-Beta Score (where the F1 score is frequently used), as well as the ROC and AUC. You can see that the \beta$\beta$ parameter controls the degree to which precision is weighed into the F score, which allows precision and recall to be considered simultaneously. The most common value for beta is 1, as this is where you are finding the harmonic average between precision and recall.

F - Beta Score

$$F_\beta = (1 + \beta^2) \frac{\text{Precision} * \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}}$$

- **ROC Curve & AUC** - By finding different thresholds for our classification metrics, we can measure the area under the curve (where the curve is known as a ROC curve). Similar to each of the other metrics above, when the AUC is higher (closer to 1), this suggests that our model performance is better than when our metric is close to 0.

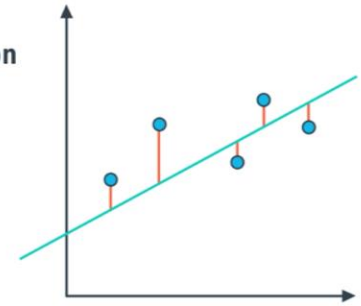


Regression Measures

You want to measure how well your algorithms are performing on predicting numeric values? In these cases, there are three main metrics that are frequently used. **mean absolute error**, **mean squared error**, and **r2** values.
As an important note, optimizing on the mean absolute error may lead to a different 'best model' than if you optimize on the mean squared error. However, optimizing on the mean squared error will **always** lead to the same 'best' model as if you were to optimize on the **r2** value.
Again, if you choose a model with the best r2 value (the highest), it will also be the model that has the lowest (MSE). Choosing one versus another is based on which one you feel most comfortable explaining to someone else.

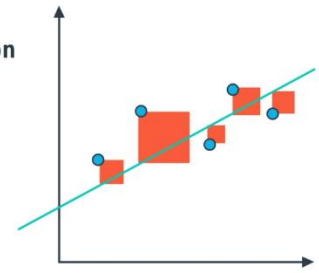
- **Mean Absolute Error (MAE) -** The first metric you saw was the mean absolute error. This is a useful metric to optimize on when the value you are trying to predict follows a skewed distribution. Optimizing on an absolute value is particularly helpful in these cases because outliers will not influence models attempting to optimize on this metric as much as if you use the mean squared error. The optimal value for this technique is the median value. When you optimize for the R2 value of the mean squared error, the optimal value is actually the mean.

```
from sklearn.metrics import mean_absolute_error
from sklearn.linear_model import LinearRegression

classifier = LinearRegression()
classifier.fit(X,y)

guesses = classifier.predict(X)

error = mean_absolute_error(y, guesses)
```



- **Mean-Squared Error (MSE)** - The mean squared error is by far the most used metric for optimization in regression problems. Similar to with MAE, you want to find a model that minimizes this value. This metric can be greatly impacted by skewed distributions and outliers. When a model is considered optimal via MAE, but not for MSE, it is useful to keep this in mind. In many cases, it is easier to actually optimize on MSE, as the a quadratic term is differentiable. However, an absolute value is not differentiable. This factor makes this metric better for gradient based optimization algorithms.

```
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression

classifier = LinearRegression()
classifier.fit(X,y)

guesses = classifier.predict(X)

error = mean_squared_error(y, guesses)
```



- **R2 Score** - Finally, the r2 value is another common metric when looking at regression values. Optimizing a model to have the lowest MSE will also optimize a model to have the the highest R2 value. This is a convenient feature of this metric. The R2 value is frequently interpreted as the 'amount of variability' captured by a model. Therefore, you can think of MSE, as the average amount you miss by across all the points, and the R2 value as the amount of the variability in the points that you capture with a model.

BAD MODEL

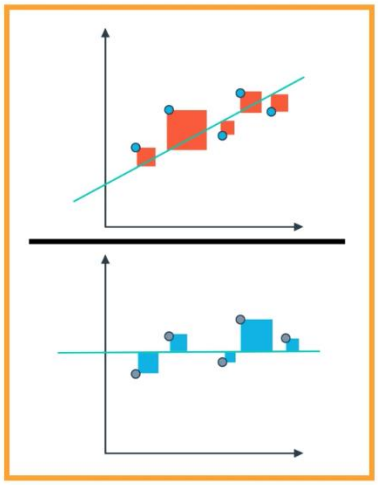
The errors should be similar.
R2 score should be close to 0.

GOOD MODEL

The mean squared error for the linear regression model should be a lot smaller than the mean squared error for the simple model.

R2 score should be close to 1.

$$R2 = 1 -$$



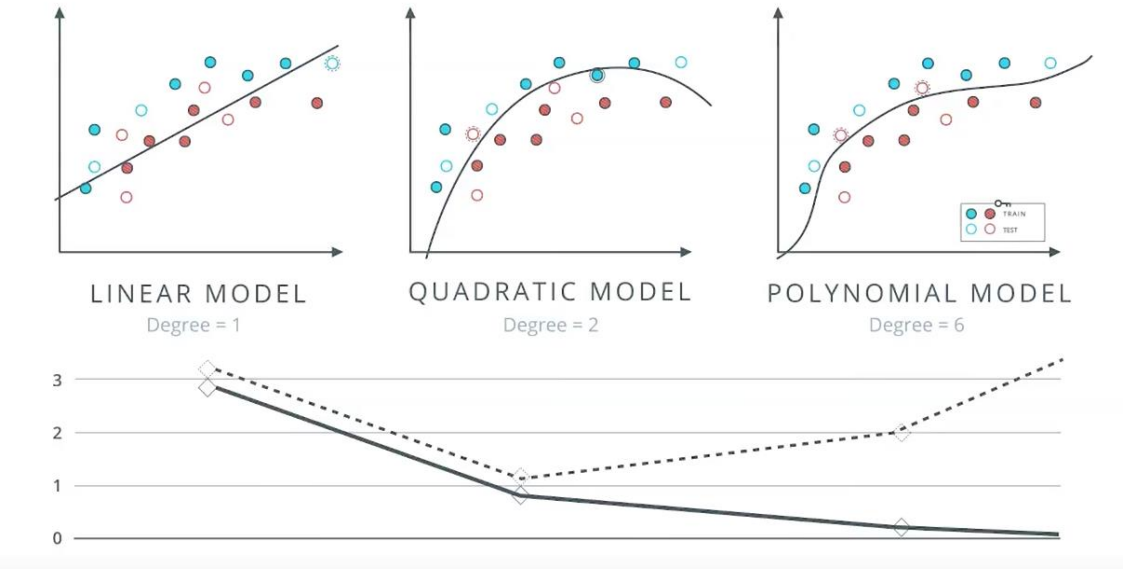
Training and Tuning

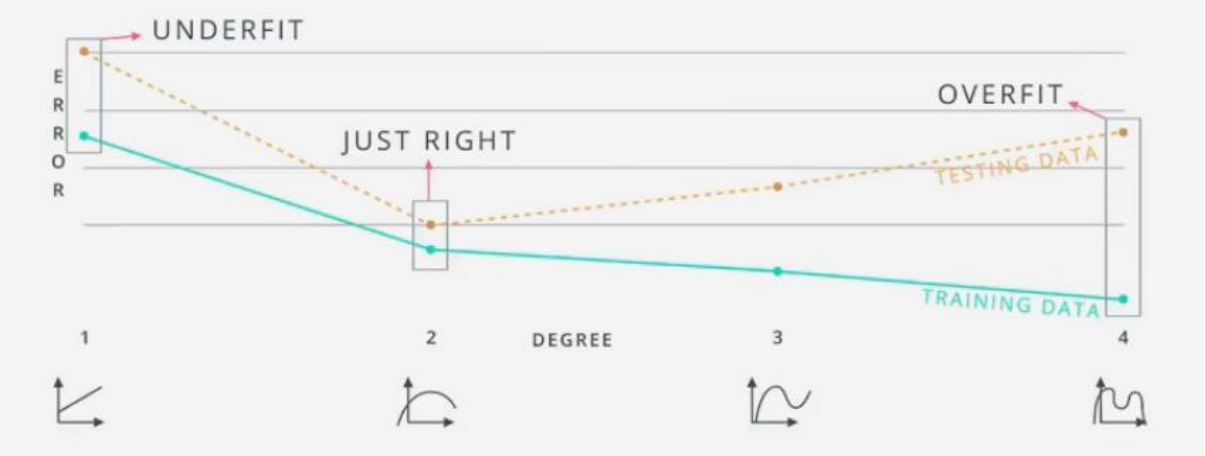
Types of error

Instead of choosing dog and not dog, if we choose animal and not animals, we miss out the cat in purple. So, model is not understanding the features properly here.

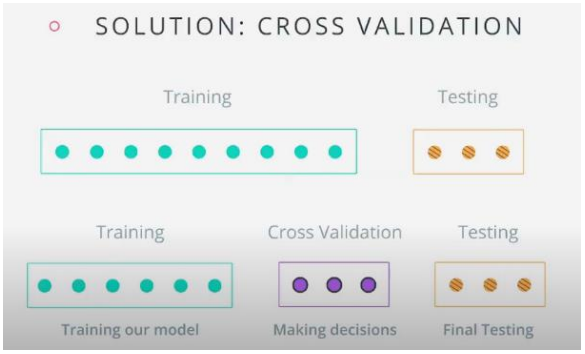
Now choosing the class as Dogs wagging tails complicates our model now.

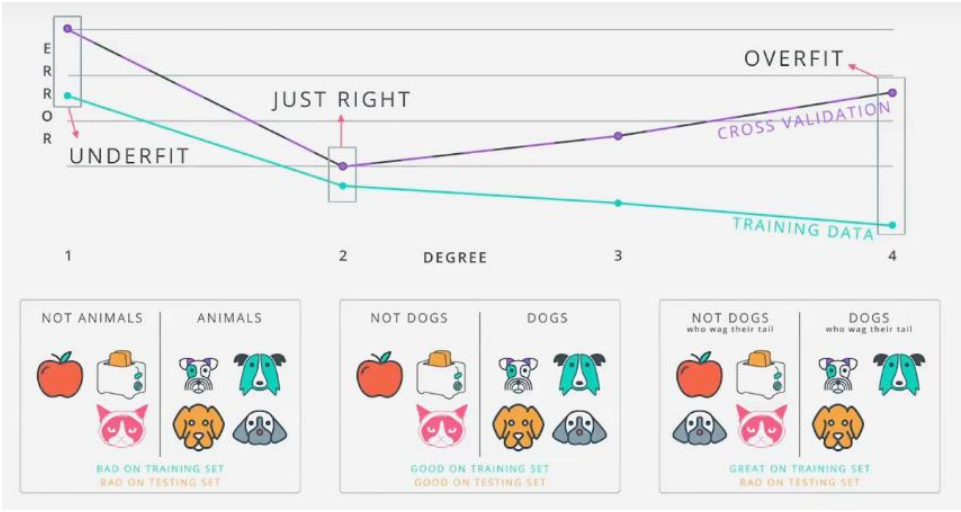
Model Complexity Graph

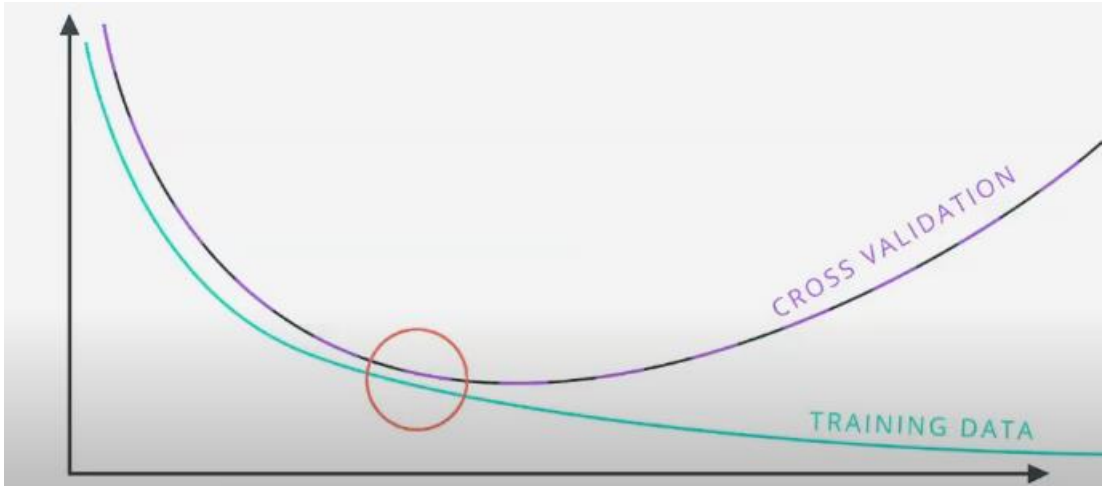




It seems wee used the test data for training above. How to deal with this problem?

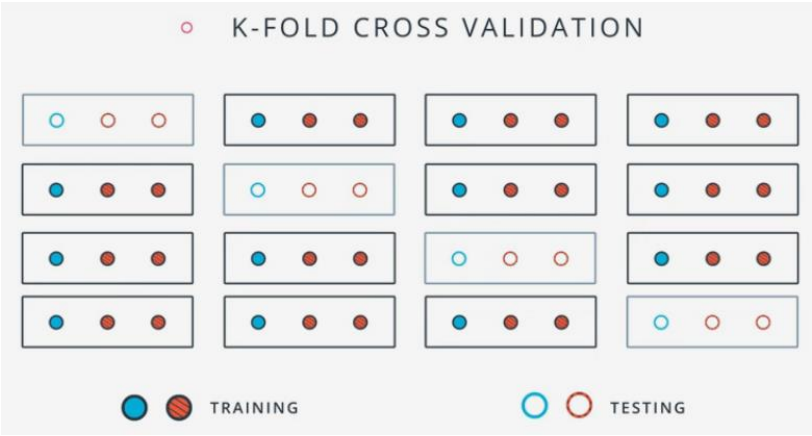

SOLUTION: CROSS VALIDATION
Training
Testing
Training
Cross Validation
Testing
Training our model
Making decisions
Final Testing

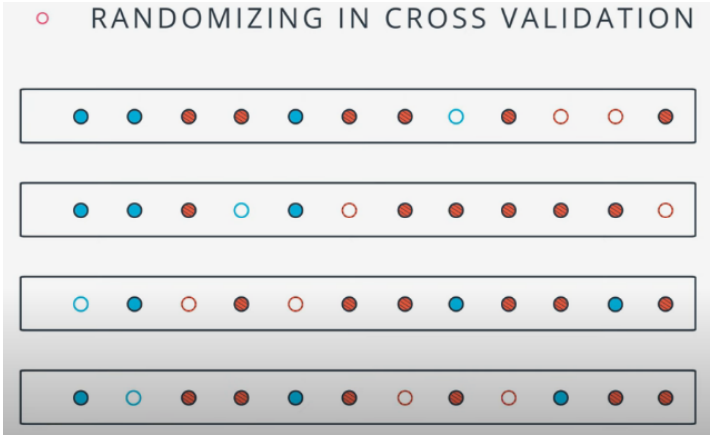

ERROR
OVERFIT
JUST RIGHT
UNDERFIT
CROSS VALIDATION
TRAINING DATA
1
2
DEGREE
3
4
NOT ANIMALS
ANIMALS
BAD ON TRAINING SET
BAD ON TESTING SET
NOT DOGS
DOGS
GOOD ON TRAINING SET
GOOD ON TESTING SET
NOT DOGS
who wag their tail
DOGS
who wag their tail
GREAT ON TRAINING SET
BAD ON TESTING SET


CROSS VALIDATION
TRAINING DATA

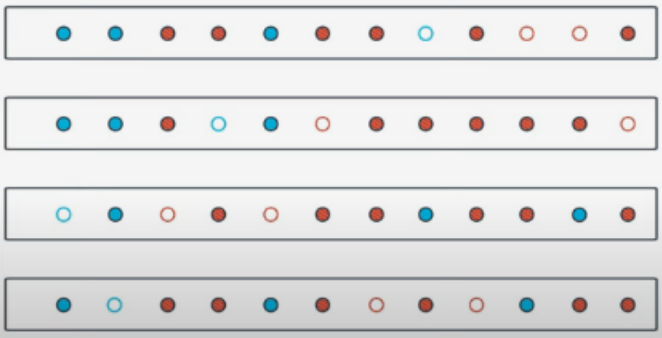
K fold Cross Validation

When we split the data in train, test we miss out the chance of utilizing complete data.



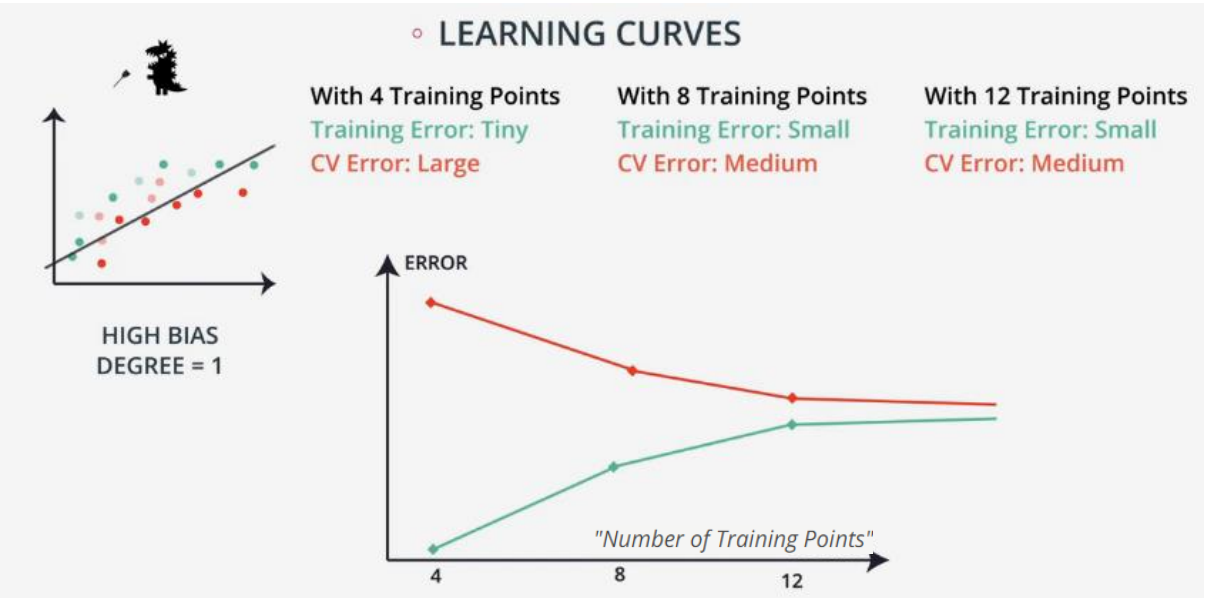



```
from sklearn.model_selection import KFold
kf = KFold(12, 3, shuffle = True)

for train_indices, test_indices in kf:
    print train_indices, test_indices

[ 0  1  2  3  4  5  6  8 11 ]  [7  9 10]
[ 0  1  2  4  6  7  8  9 10 ]  [3  5 11]
[ 1  3  5  6  7  8  9 10 11]   [0  2  4]
[ 0  2  3  4  5  7  9 10 11]   [1  6  8]
```

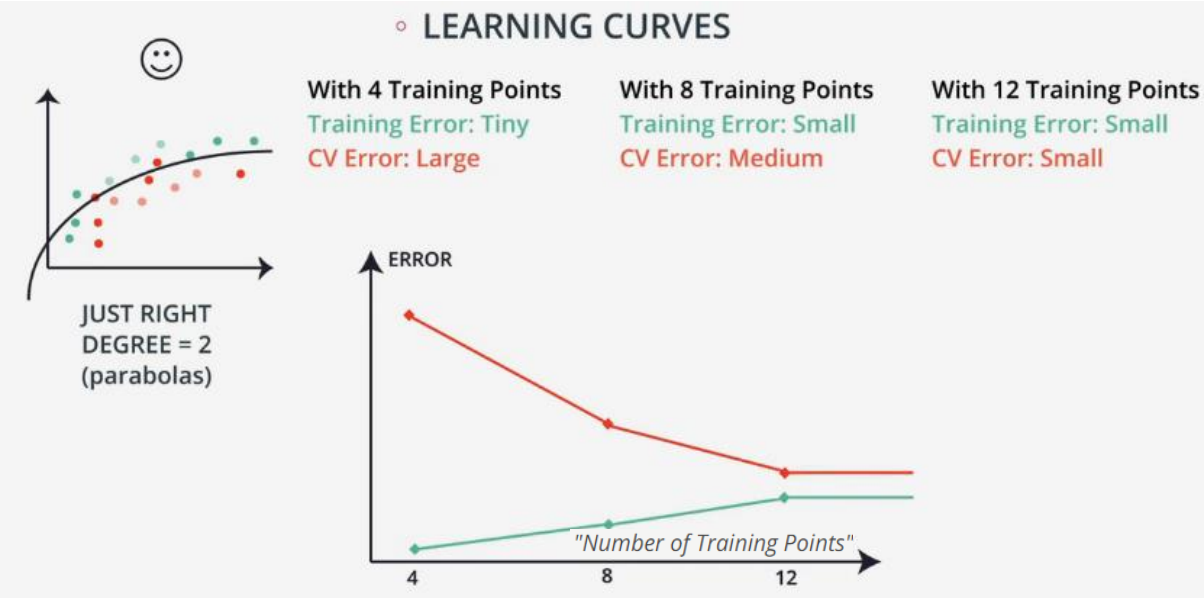


Learning Curves

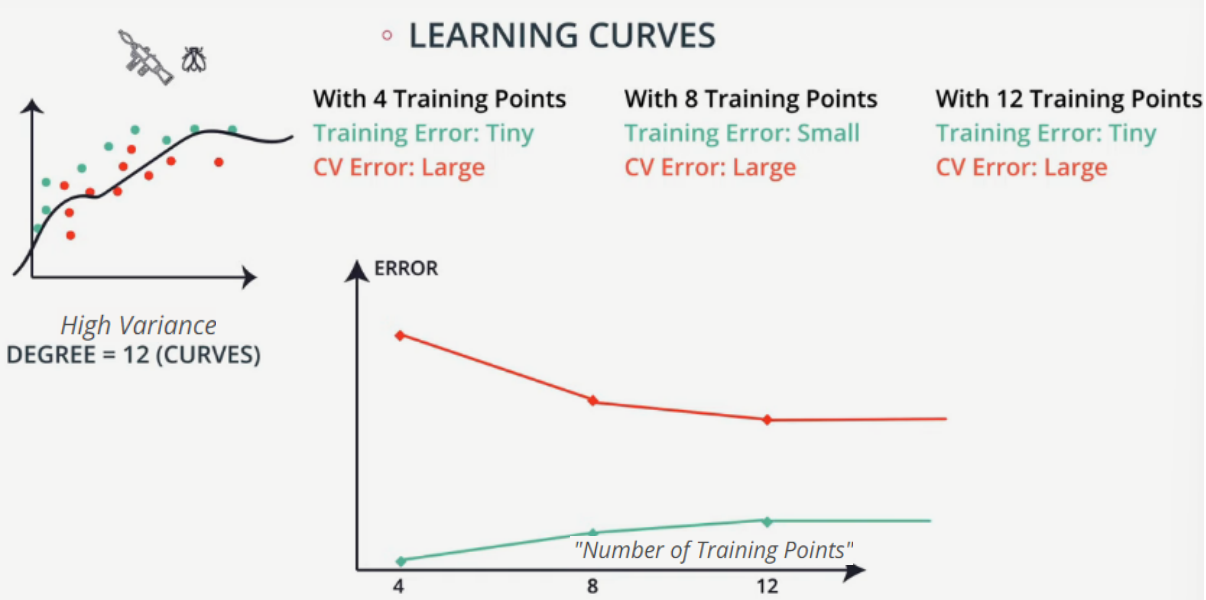
In case of linear model:



In case of quadratic model:



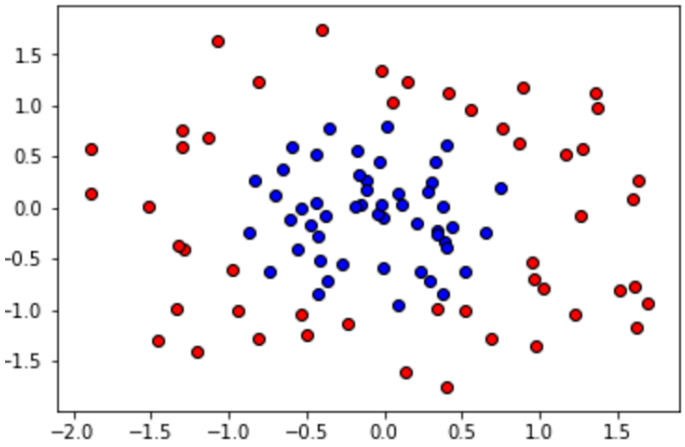
For degree 6 model:



Detecting Overfitting and Underfitting with Learning Curves

In this example., we'll be using three models to train the circular dataset below.

- A Decision Tree model,
- a Logistic Regression model, and
- a Support Vector Machine model.



One of the models overfits, one underfits, and the other one is just right.

We'll be using the function called `learning_curve`:

```
•       train_sizes, train_scores, test_scores = learning_curve(
•           estimator, X, y, cv=None, n_jobs=1, train_sizes=np.linspace(.1, 1.0,
num_trainings))
```

No need to worry about all the parameters of this function (you can read some more in **here**, but here we'll explain the main ones:

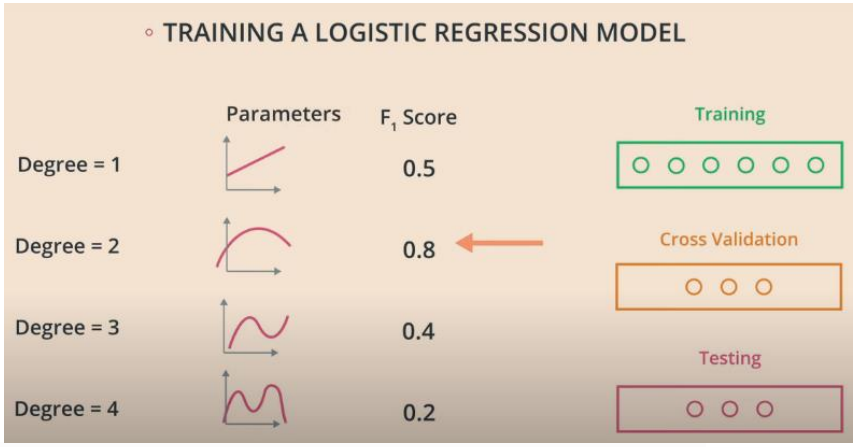
- `estimator`, is the actual classifier we're using for the data, e.g., `LogisticRegression()` or `GradientBoostingClassifier()`.
- `X` and `y` is our data, split into features and labels.
- `train_sizes` are the sizes of the chunks of data used to draw each point in the curve.
- `train_scores` are the training scores for the algorithm trained on each chunk of data.
- `test_scores` are the testing scores for the algorithm trained on each chunk of data.

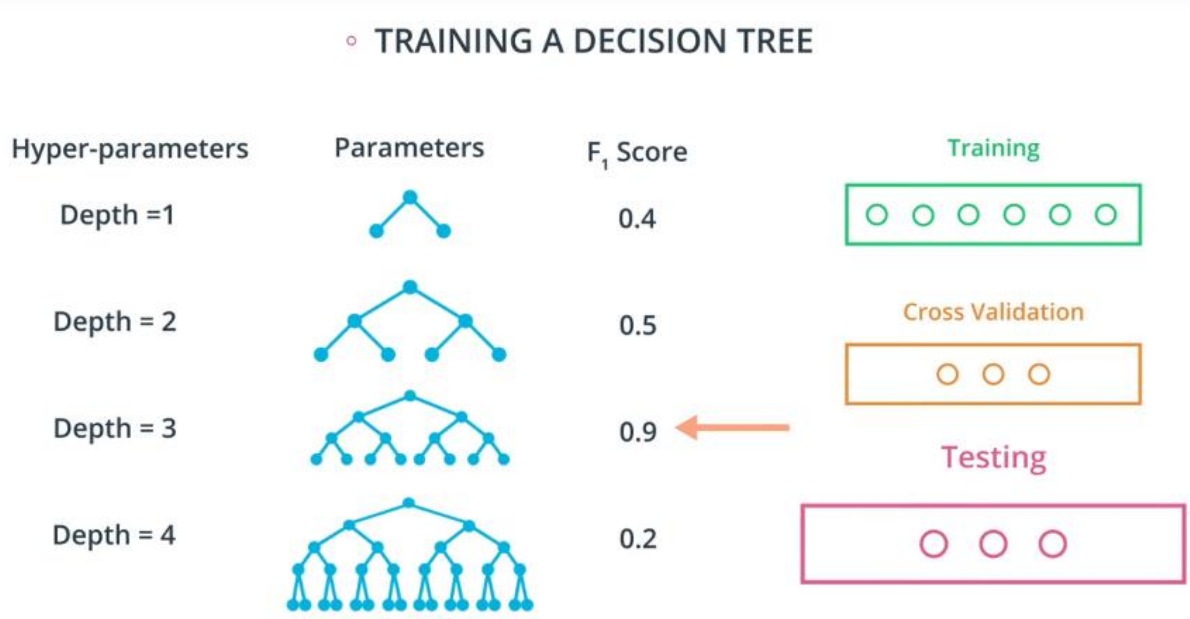
Two very important observations:

- The training and testing scores come in as a list of 3 values, and this is because the function uses 3-Fold Cross-Validation.
- **Very important:** As you can see, we defined our curves with Training and Testing **Error**, and this function defines them with Training and Testing **Score**. These are opposite, so the higher the error, the lower the score. Thus, when you see the curve, you need to flip it upside down in your mind, in order to compare it with the curves above.

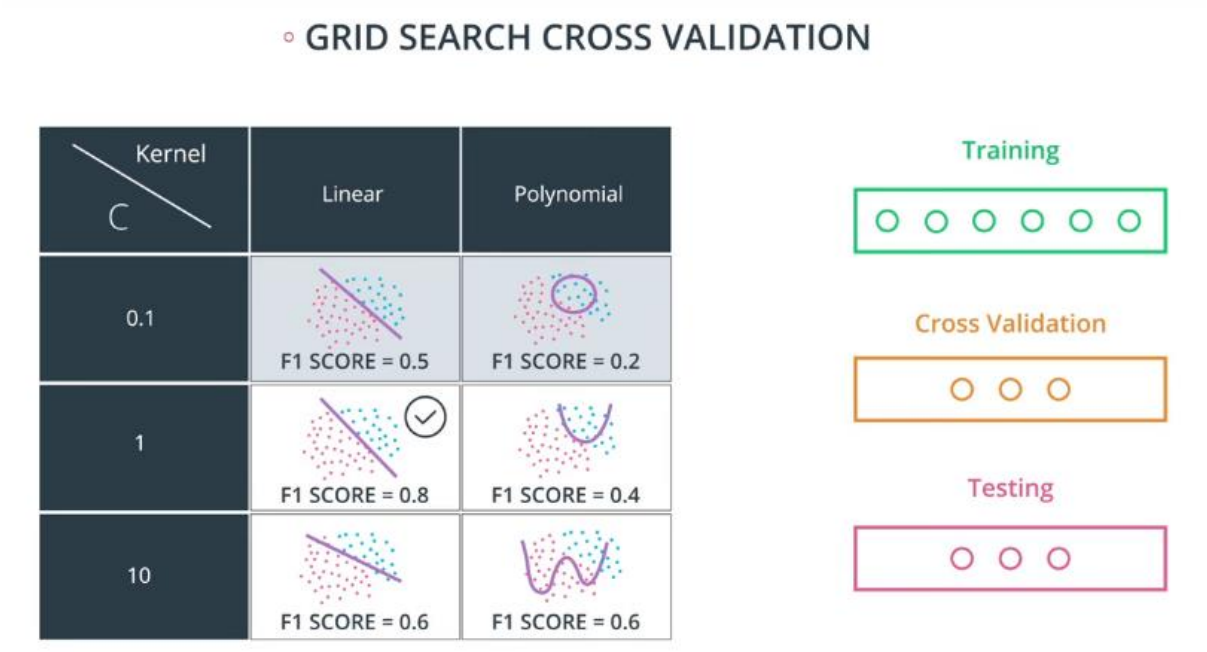
Grid Search

We choose the model with degree that has highest F1 score. And verify it with testing data.





In case of a greater number of hyperparameters, how do we select the best model?



Grid Search in sklearn

Let's say we'd like to train a support vector machine, and we'd like to decide between the following parameters:

- kernel: `poly` or `rbf`.
- C: 0.1, 1, or 10.

The steps are the following:

1. Import GridSearchCV

```
from sklearn.model_selection import GridSearchCV
```

2. Select the parameters:

Here we pick what are the parameters we want to choose from, and form a dictionary. In this dictionary, the keys will be the names of the parameters, and the values will be the lists of possible values for each parameter.

```
parameters = {'kernel':['poly', 'rbf'],'C':[0.1, 1, 10]}
```

3. Create a scorer.

We need to decide what metric we'll use to score each of the candidate models. In here, we'll use F1 Score.

```
from sklearn.metrics import make_scorer
from sklearn.metrics import f1_score
scorer = make_scorer(f1_score)
```

4. Create a GridSearch Object with the parameters, and the scorer. Use this object to fit the data.

```
        # Create the object.
        grid_obj = GridSearchCV(clf, parameters, scoring=scorer)
        # Fit the data
        grid_fit = grid_obj.fit(X, y)
```

5. Get the best estimator.

```
        best_clf = grid_fit.best_estimator_

```

Now you can use this estimator `best_clf` to make the predictions.