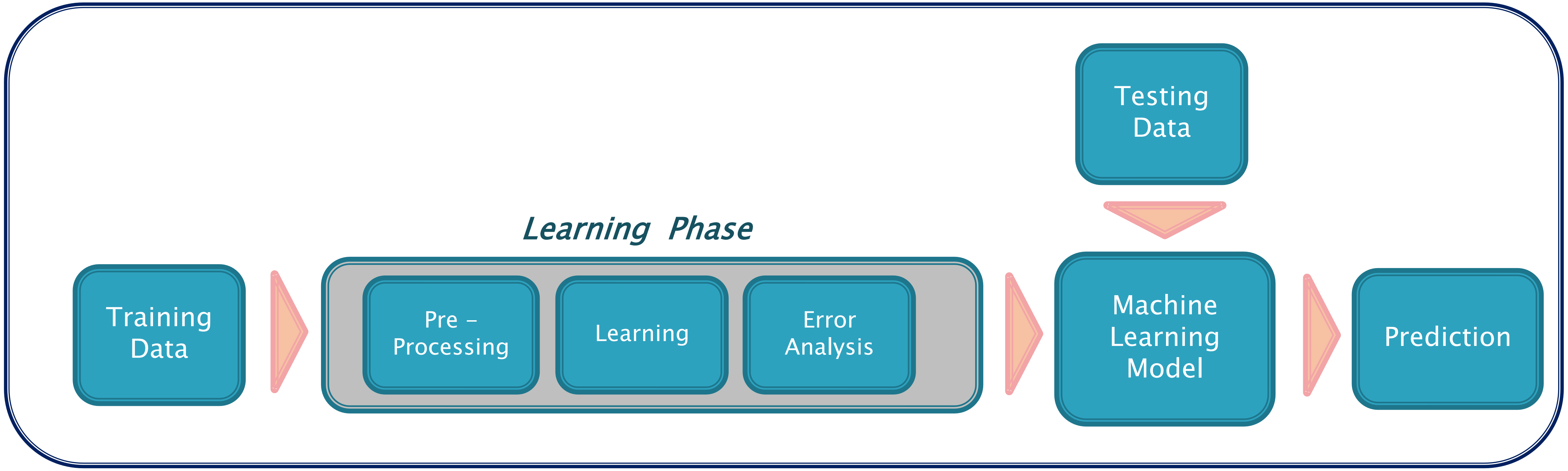
Machine Learning with Scikit Learn

Pratap Dangeti

Overview of Machine Learning

Machine learning algorithms are computer system that can adapt and learn from their experience



Two of the most widely adopted machine learning methods are

- Supervised learning are trained using labeled examples, such as an input where the desired output is known e.g. regression or classification
- Unsupervised learning is used against data that has no historical labels e.g. cluster analysis
- Third ML paradigm is Semi-supervised learning which is used when there are strong reasons to believe that a typical pattern exists in data such that the given pattern can be quantified via models.

Machine Learning vs. Statistical Modeling

Machine Learning is

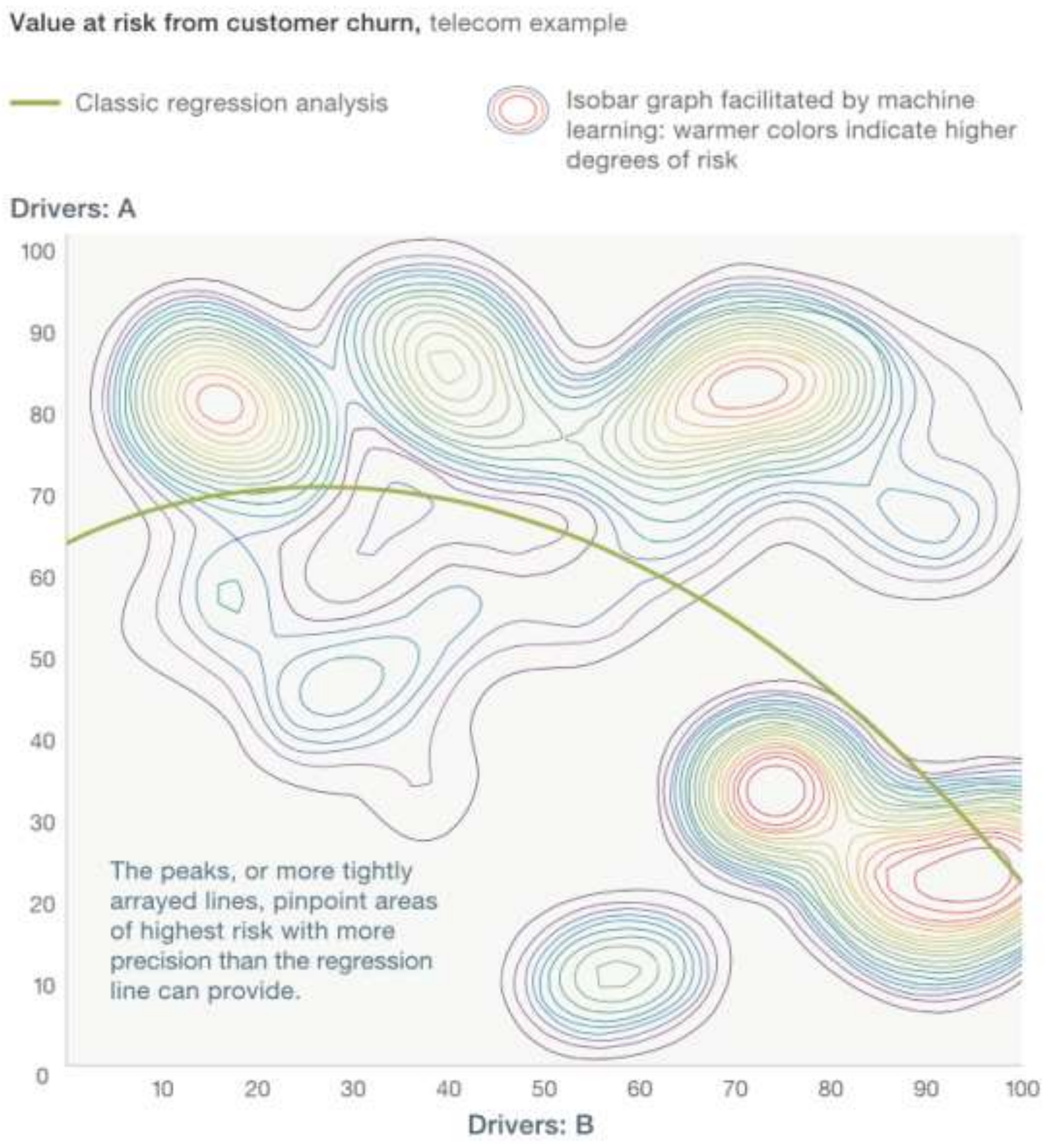
- Algorithm that can learn from the data without relying on rules-based programming
- E.g.: Machine Learning predicts the output with the accuracy of 85 %
- Machine Learning is from the school of computer science

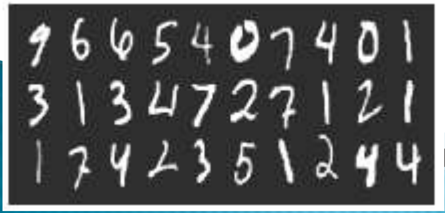
Statistical Modeling is

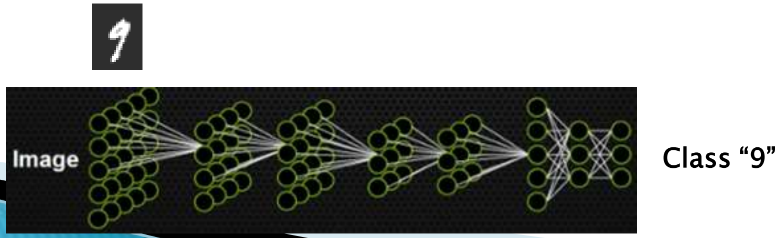
- Formalization of relationships between variables in the form of mathematical equations
- E.g.: Statistical model predicts the output with the accuracy of 85 % with 90% confidence
- Statistical Modeling is from the school of Statistics & Mathematics

Digit Recognizer

- Hand written digits cannot be modeled mathematically using equations. Machine learning models, trained with thousands of examples classify surprisingly







Bias vs. Variance Tradeoff

Bias vs. Variance Tradeoff

- High variance model will tend to vary model's estimate considerably even to the small change in data points

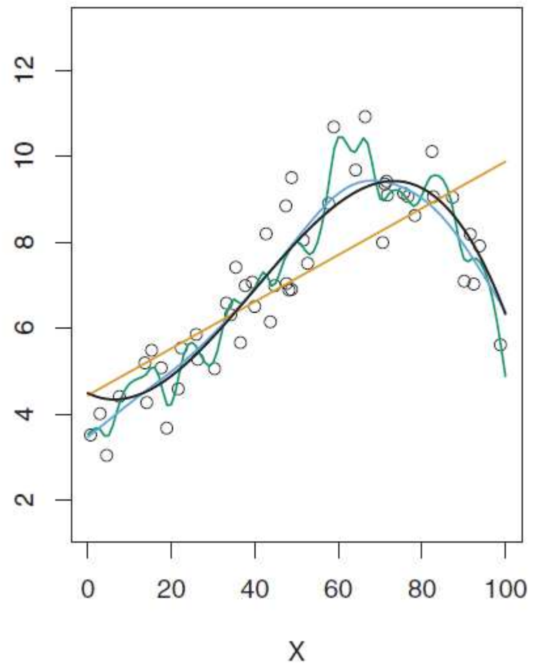
- On the other hand high bias models are robust enough and do not change estimate much for the change in data points

Over fitting vs. Under fitting

- High variance models (usually low bias) over fits the data
- Low variance models (usually high bias) under fits the data

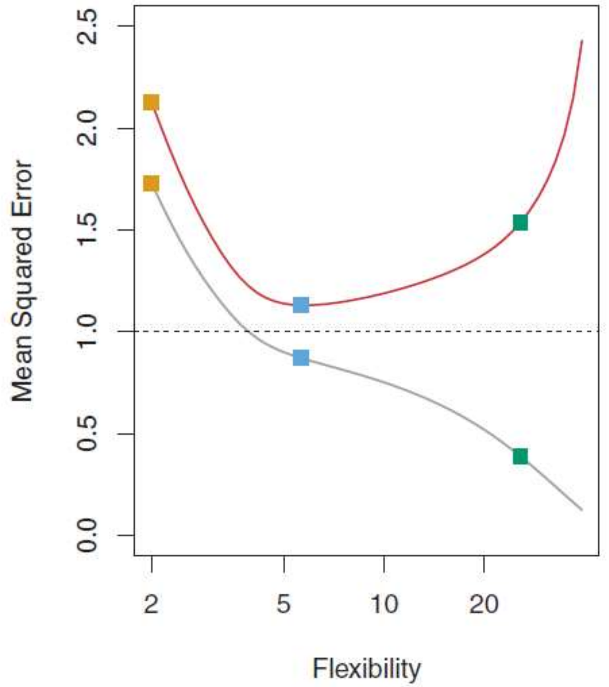
Ideal model will have both low bias & low variance

Tip:
1] If your model has high bias then adding more features will work (going from model degree 1 to degree 2 etc.)
2] If your model has high variance, remove features (from degree 2 to degree 1) or try to add more data will work



Bias vs. Variance Tradeoff of Model comparison with various degrees of non linearity on Train Data

Error comparison of Train & Test datasets based on varied degree of non linearity



Model underfits if both Train & Test errors are high

Effect of more data points on non linear models

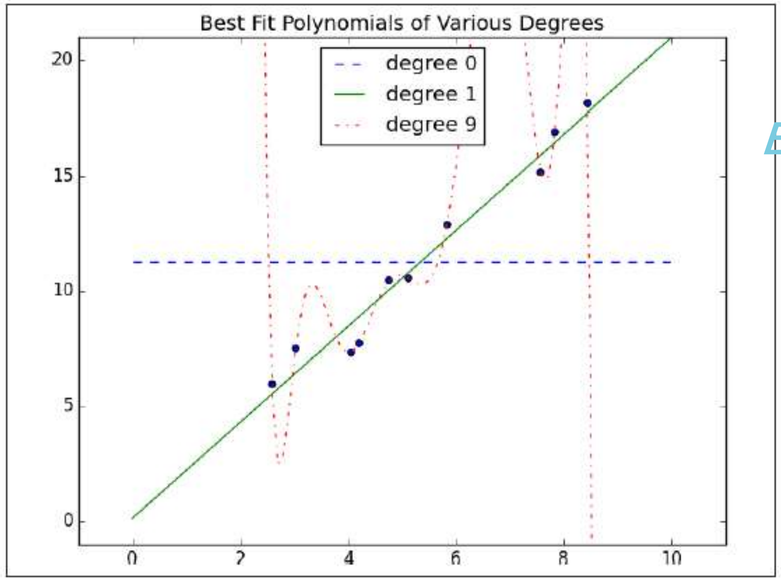
Effect of more data

- Fitting model with degree 9 on different sample sizes, it can be observed that if we train on 100 data points instead 10 data points there would be less issue of overfitting

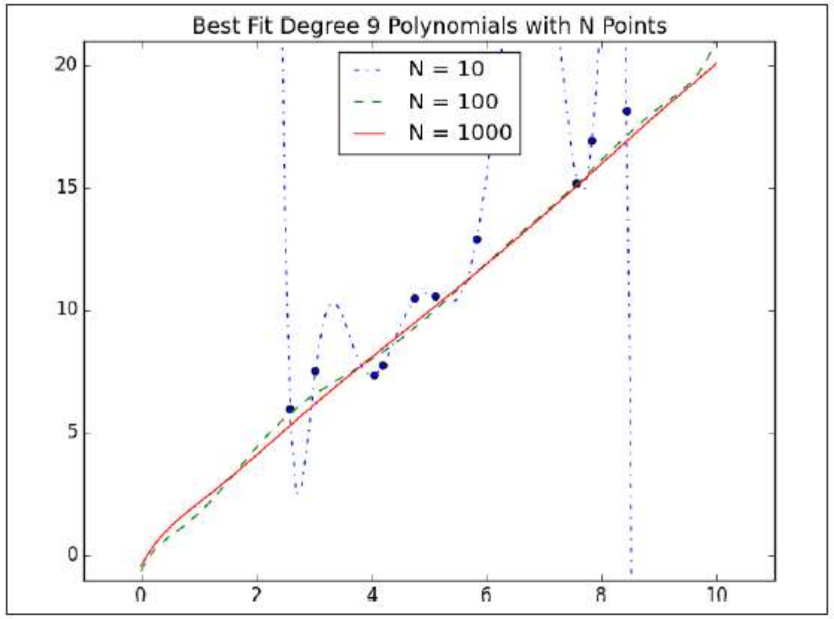
- Model trained from 1000 data points look very similar to the degree 1 model on small data of 10 data points

Holding model complexity constant, the more data you have, the harder it is to overfit

On the other hand, more data won't help model with high bias



Effect of model non linearity on Small data



Effect of model non linearity with Degree 9 on increasing samples of data

Training Validation & Testing

Statistical Modeling Methodology

- Training data used to train the model
- Testing data used to test the accuracy of the model

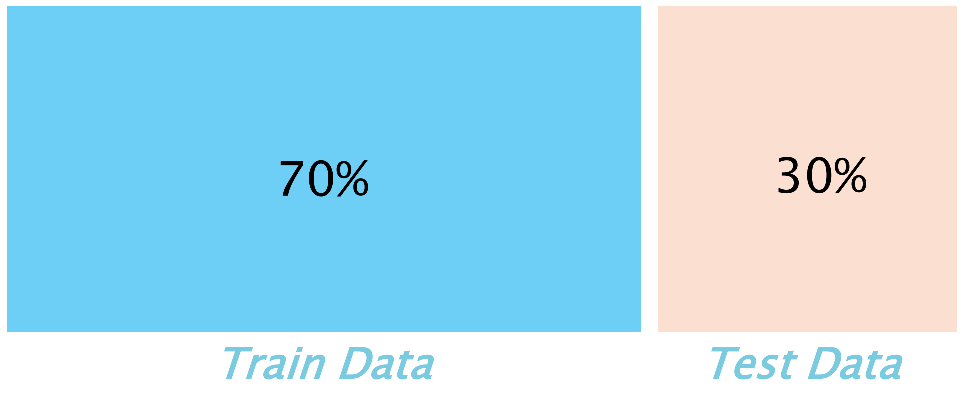
Machine Learning Modeling Methodology

- Train data used to train model by pairing input with expected output
- Validation data used to check how well model has been trained and to estimate model properties (mean error, classification error, precision, recall etc.)
- Finally calculate accuracy on Test data

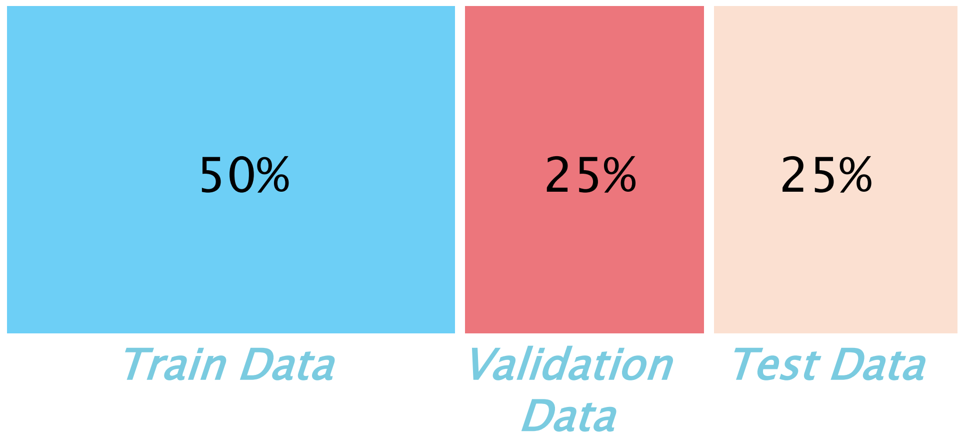
- *In first part you look at your model and select the best performing approach using the validation data*
- *Then you estimate the accuracy of the model approach based on test data*

Why separate Validation & Test data sets required ?
The error rate estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is used to select the final model After assessing the final model on the test set, YOU MUST NOT tune the model any further!

Statistical Modeling Methodology



Machine Learning Modeling Methodology

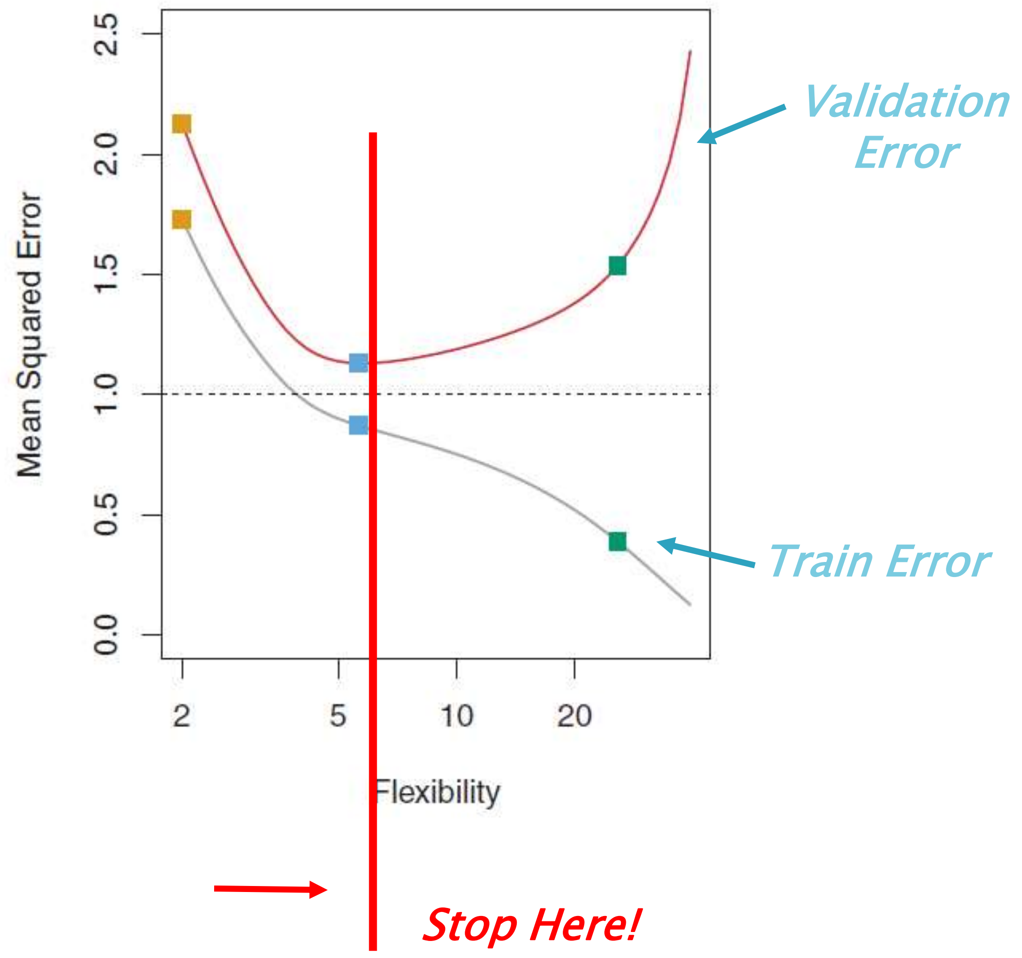


When To Stop Tuning Model ?

Machine Learning Model Tuning

- Always keep a tab on train & validation errors while tuning the algorithm
- Stop increasing flexibility/degrees of the model when validation error starts increasing

Tuning of Machine Learning Models



List of Machine Learning Algorithms

Supervised Learning

Classification/Regression

- Linear Regression
- Polynomial Regression
- Logistic Regression
- Decision Trees
- Random Forest
- Boosting
- Support Vector Machines
- KNN (K-Nearest Neighbors)
- Neural Networks
- Naïve Bayes

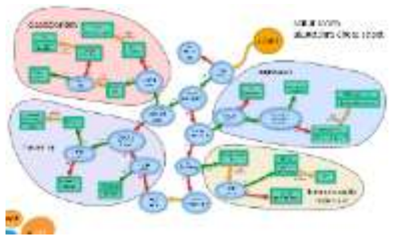
Unsupervised Learning

Clustering & Variable reduction

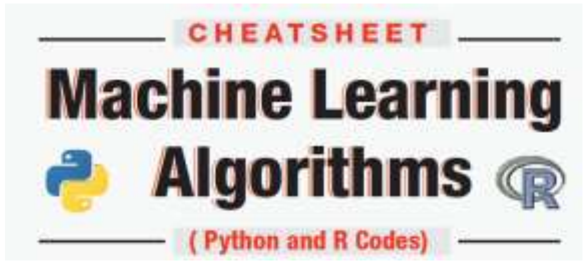
- K-means clustering
- PCA (Principal Component Analysis)

Supporting Techniques

- Cross Validation
- Gradient Descent
- Grid Search



ML Cheatsheet



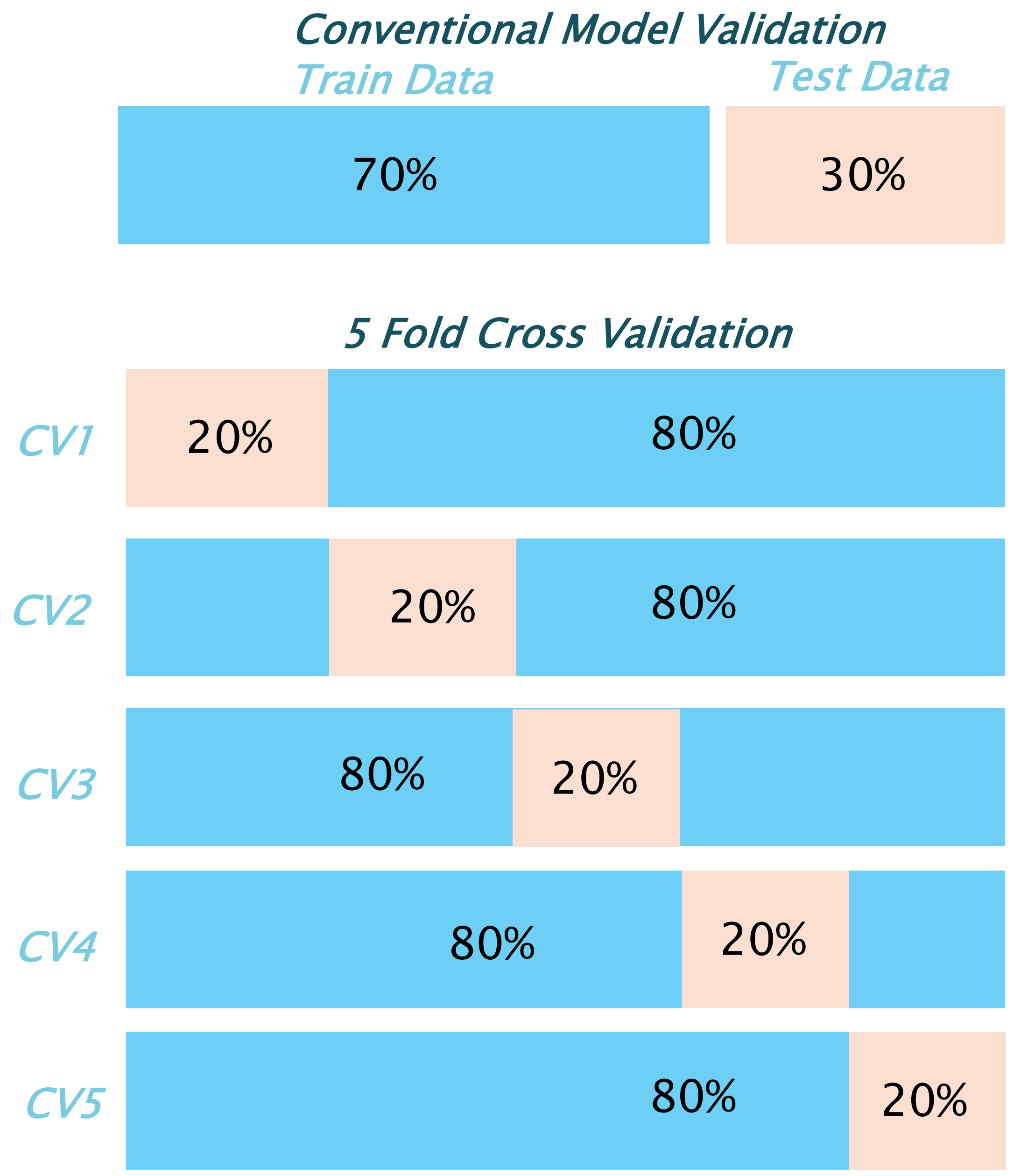
ML Algorithms & Codes

Cross Validation

Cross Validation

- Cross validation improves the robustness of the models & provides the mean errors by averaging all possibilities as the models covers entire data points within with mix & match

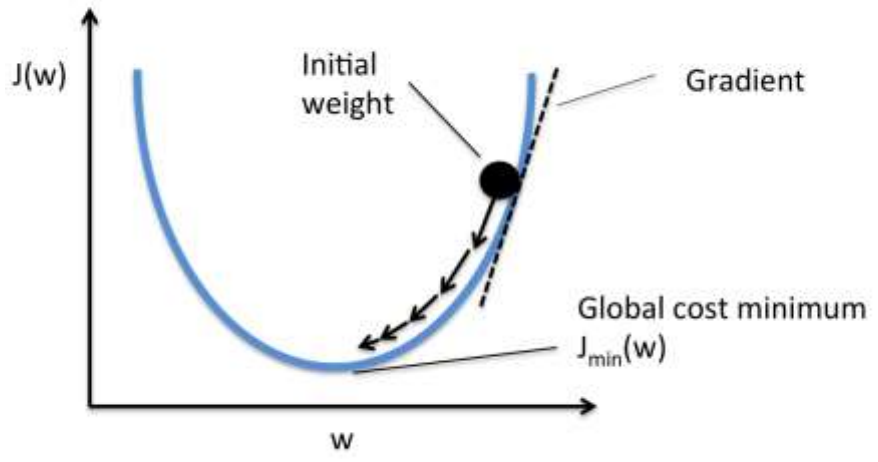
5 Fold CV Error
= (CV1 Error + CV2 Error+ CV3 Error+ CV4 Error+ CV5 Error) / 5



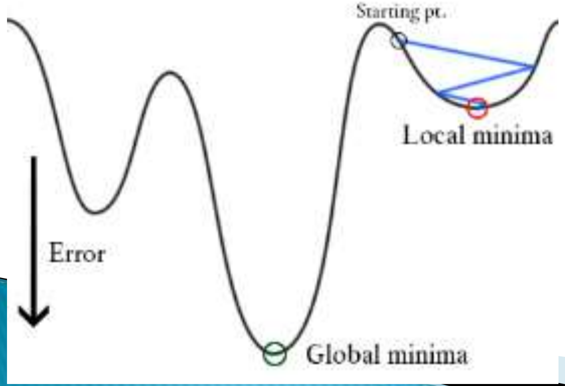
Gradient Descent

- **Gradient Descent (SGD):** Gradient descent is a way to minimize an objective function **J(θ)** parameterized by a model's parameter **$\theta \in R^d$** by updating the parameters in the opposite direction of the gradient of the objective function w.r.to the parameters. Learning rate determines the size of steps taken to reach minimum.
  - Batch Gradient Descent (all training observations per each iteration)
  - SGD (1 observation per iteration)
  - Mini Batch Gradient Descent (size of about 50 training observations for each iteration)

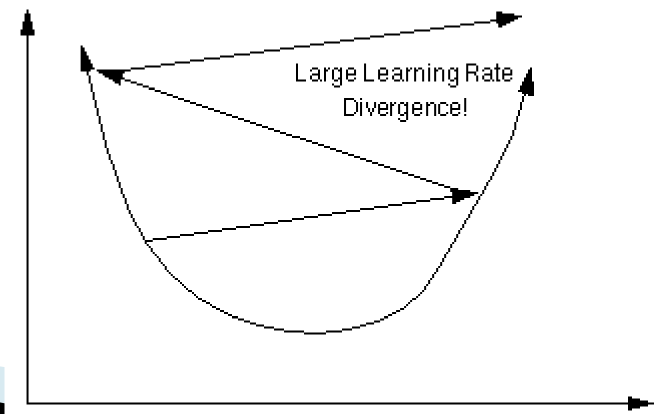
Gradient Descent



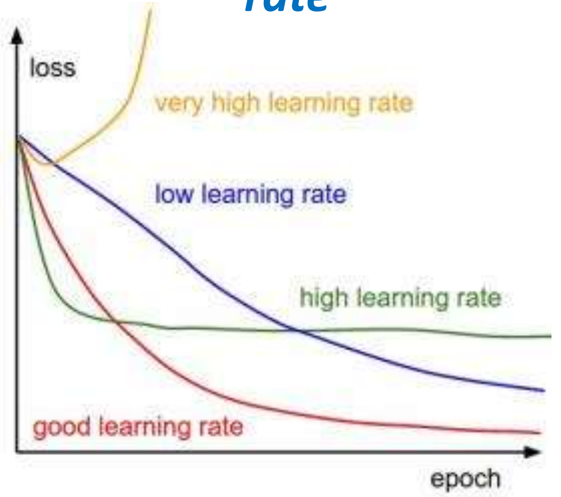
Non Convex Function



High Learning Rate cause Divergence



Loss minimization w.r.to Learning rate



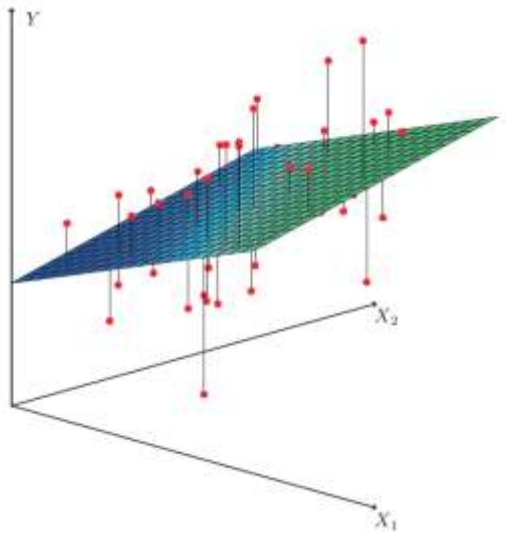
Linear Regression

Statistical Multiple Linear Regression Assumptions

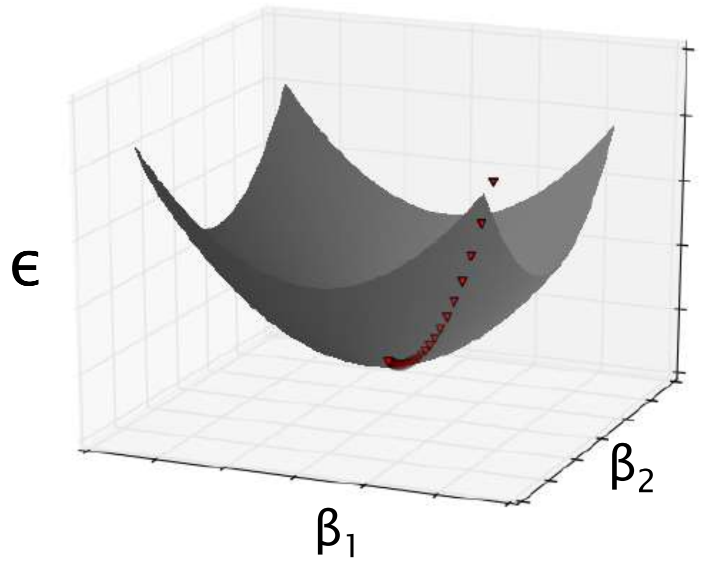
- Independent variable Y should be a linear combination of dependent variables (X1, X2 …)
- Multivariate normality
- No or little multi-collinearity
- No auto-correlation
- Homoscedasticity (Errors should have constant variance)

For Machine learning no assumptions are required, if model fits well, it should be able to generate high accuracy

Statistical way



Machine learning way

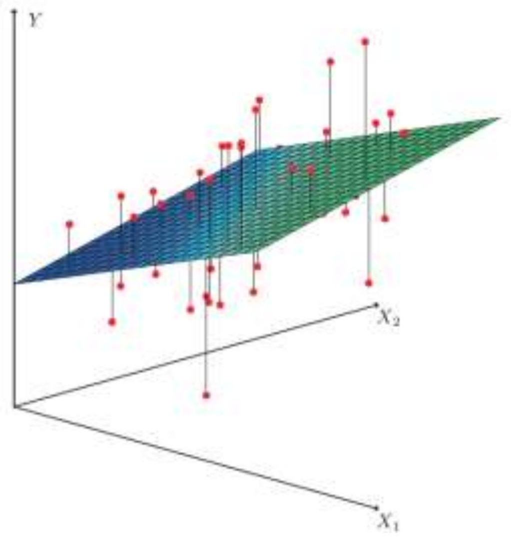


$$\hat{Y} = \beta_1 * X_1 + \beta_2 * X_2$$

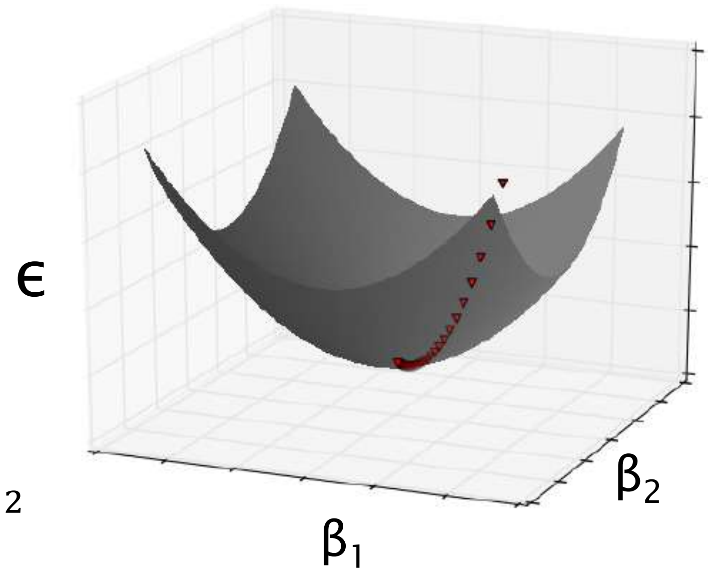
$$\varepsilon = (Y - (\beta_1 * X_1 + \beta_2 * X_2))^2$$

Linear Regression

Statistical way



Machine learning way



$$\hat{Y} = \beta_1 * X_1 + \beta_2 * X_2$$

$$\varepsilon = (Y - (\beta_1 * X_1 + \beta_2 * X_2))^2$$

```
import matplotlib.pyplot as plt , import numpy as np
from sklearn import linear_model
x_train,x_test,y_train,y_test =
train_test_split( data.data, data.target )

regr = linear_model.LinearRegression ( )
regr.fit ( x_train, y_train )
print("Mean squared error: %.2f" %
np.mean (( regr.predict ( x_test ) - y_test ) ** 2))
```

```
from sklearn.linear_model import SGDRegressor
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import train_test_split
x_train,x_test,y_train,y_test =
train_test_split( data.data, data.target )

regressor = SGDRegressor(loss= ‘squared_loss‘ )
regressor.fit ( x_train, y_train )
regressor.predict ( x_test )
print("Mean squared error: %.2f" %
np.mean ((regressor.predict ( x_test ) - y_test ) ** 2))
```

Various Losses in Machine Learning

Surrogate Losses in place of 0-1 Loss

- 0–1 Loss is not differentiable, hence approximated losses are being using in place
- Squared loss (For regression)
- Hinge Loss (SVM)
- Logistic Loss/ Log Loss (Logistic Regression)

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$$

$$\text{Loss}_{0\text{-}1}(x, y, \mathbf{w}) = \mathbf{1}[f_{\mathbf{w}}(x) \neq y]$$

$$= \mathbf{1}[\underbrace{(\mathbf{w} \cdot \phi(x))y}_{\text{margin}} \leq 0]$$

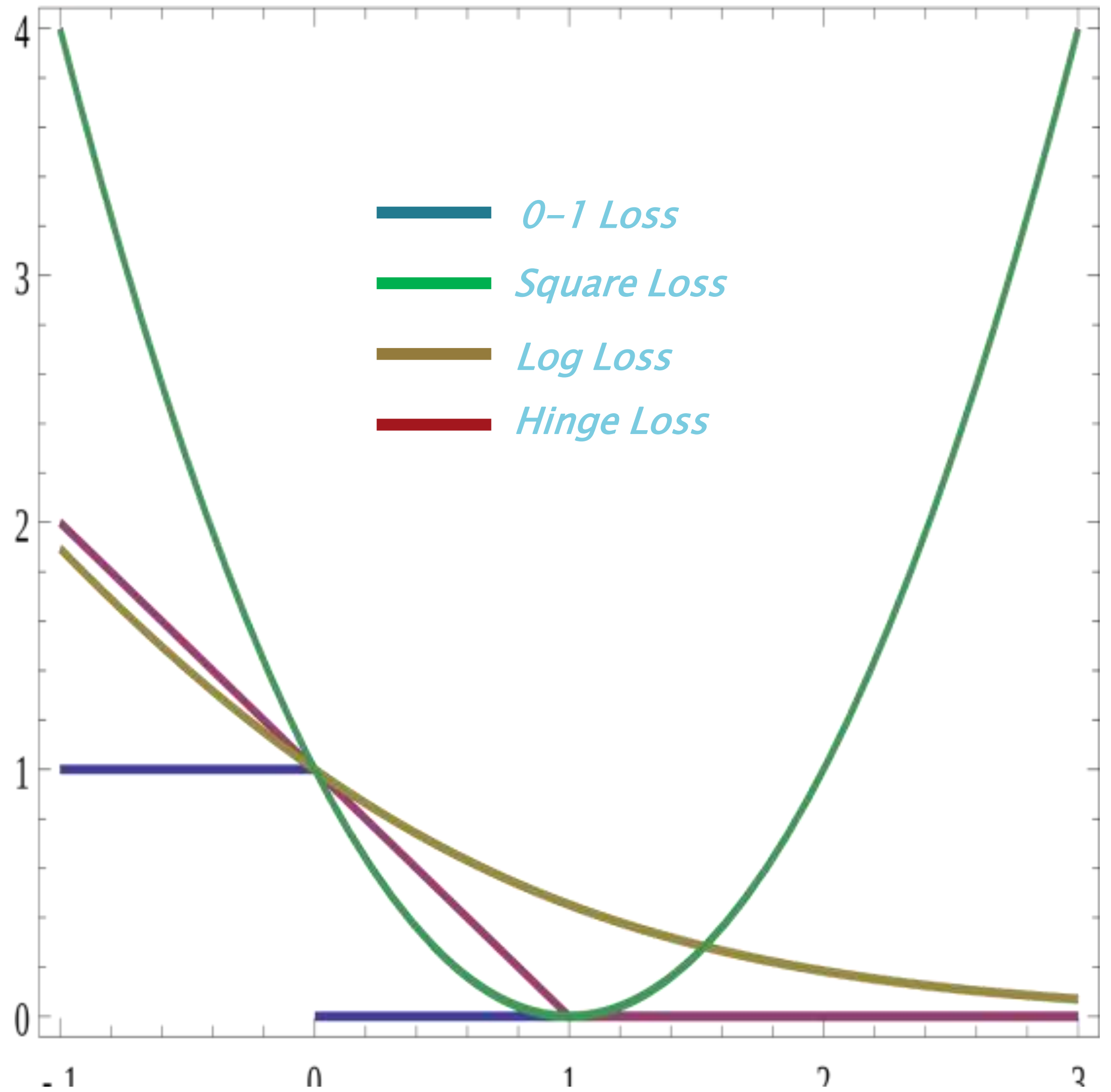
$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\underbrace{f_{\mathbf{w}}(x) - y}_{\text{residual}})^2$$

$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

$$\text{Loss}_{\text{logistic}}(x, y, \mathbf{w}) = \log(1 + e^{-(\mathbf{w} \cdot \phi(x))y})$$

*Note that all surrogates give a loss penalty of 1 for y*f(x) = 0*

Loss Functions in Machine Learning Models



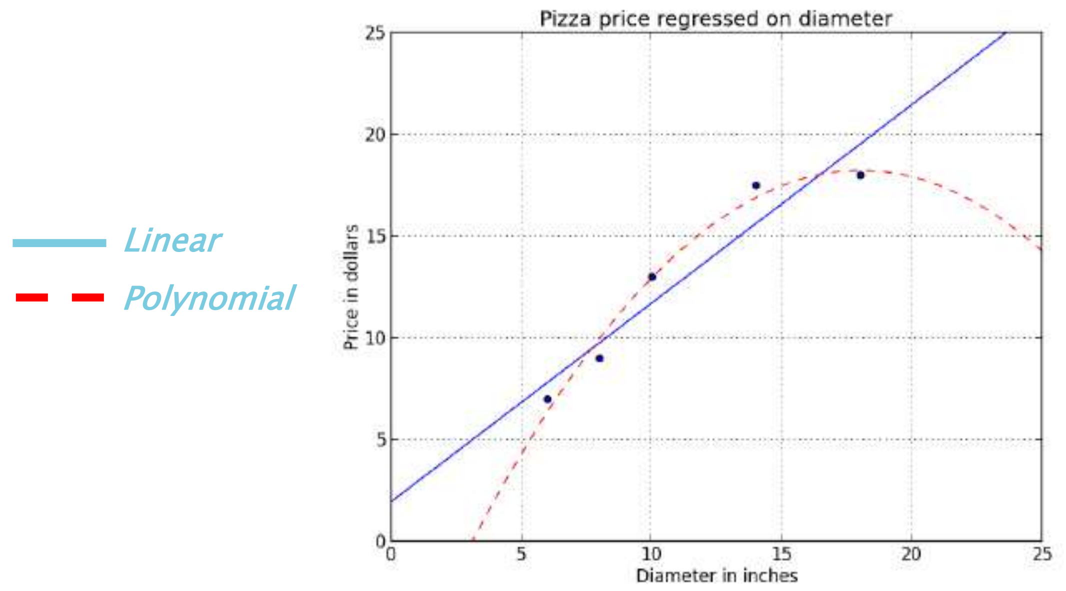
Polynomial Regression

Polynomial Regression

- Polynomial Regression is a special case of Linear Regression that adds terms with degrees greater than one to the model
- Real-world curvilinear relationship is captured when you transform the training data by adding polynomial terms, which are then fit in the same manner as in multiple linear regression

$$y = \alpha + \beta_1 x + \beta_2 x^2$$

Polynomial Regression



```
from sklearn.linear_model import Linear_Regression

regressor = LinearRegression ( )
regressor.fit( x_train, y_train)
print("Mean squared error: %.2f" %
np.mean ((regressor.predict ( x_test ) - y_test ) ** 2))
```

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Linear_Regression

quadratic_featurizer = PolynomialFeatures (degree = 2 )
x_train_quadratic = quadratic_featurizer.fit_transform( x_train )
x_test_quadratic = quadratic_featurizer.fit_transform( x_test )

regressor_quadratic = LinearRegression ( )
regressor_quadratic.fit(x_train_quadratic , y_train)

print("Mean squared error: %.2f" %
np.mean ((regressor_quadratic.predict (x_test_quadratic ) - y_test ) ** 2))
```

Logistic Regression

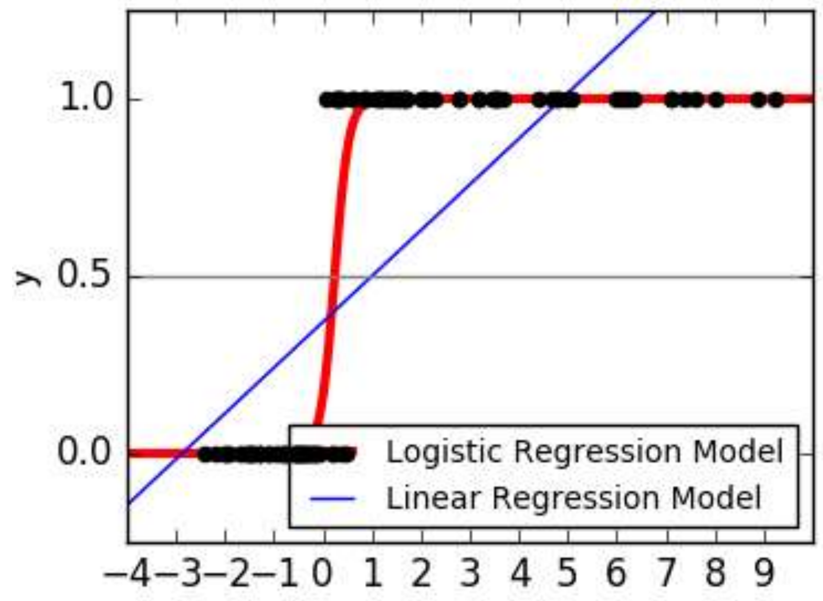
Logistic Regression

- In logistic regression response variable describes the probability that the outcome is the positive case. If the response variable is equal to or exceeds a discrimination threshold, the positive class is predicted; otherwise, the negative class is predicted
- Response variable is modeled as a function of a linear combination of the explanatory variables using the logistic function.

$$F(t) = \frac{1}{1+e^{-(\beta_0+\beta_x)}}$$

```
from sklearn.linear_model.logistic import LogisticRegression

classifier = LogisticRegression()
classifier.fit (x_train,y_train)
prediction_probabilities = classifier.predict_proba (x_test)
prediction_class = classifier.predict (x_test)
```

Logistic vs. Linear Regression Model on Binary data



```
from sklearn.linear_model import SGDClassifier
from sklearn.cross_validation import train_test_split
x_train,x_test,y_train,y_test =
train_test_split( data.data, data.target )

classifier = SGDClassifier (loss= ‘log‘ )
classifier.fit ( x_train, y_train )
prediction_probabilities = classifier.predict_proba (x_test)
prediction_class = classifier.predict (x_test)
```

Logistic Regression

ROC Curve

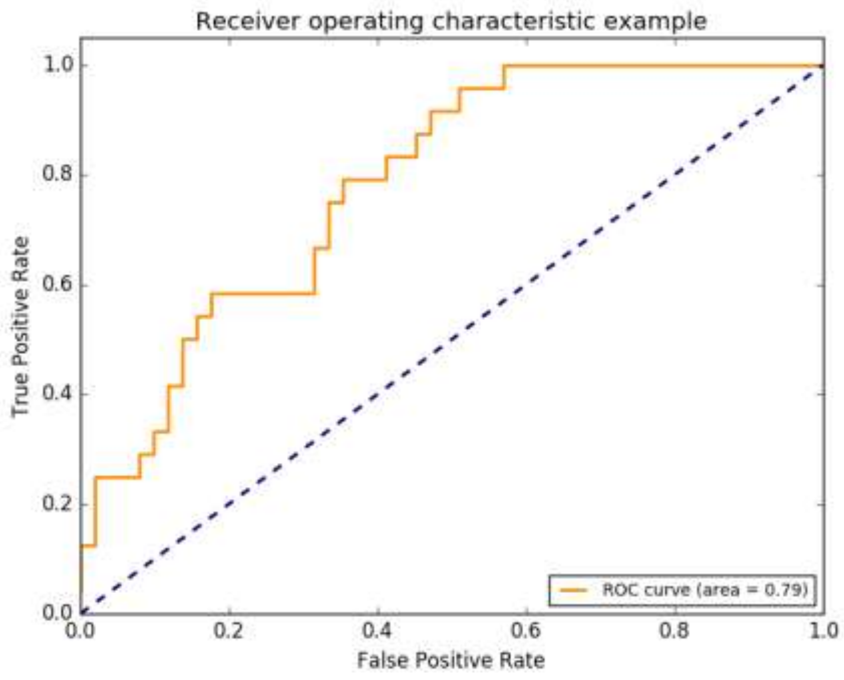
- ROC curves provide the goodness of model, higher the area under curve, the better the model is

	Predicted	
Actual	1 (Yes)	0 (No)
1 (Yes)	TP	FN
0 (No)	FP	TN

Accuracy = (TP+TN) / N
Precision = TP / (TP+FP)
Recall = TP / (TP+FN)
F1 score = 2 * P * R / (P+R)

Sensitivity = TPR = Recall = TP / (TP+FN)
1- Specifity = FPR = FP / (FP+TN)

ROC Curve



```
from sklearn.metrics import roc_curve,auc
from sklearn.cross_validation import cross_val_score

classifier = LogisticRegression()
classifier.fit (x_train,y_train)

tscores = cross_val_score( classifier, x_test, y_test, cv=5)
Print ('Test Accuracy :',np.mean(tscores),tscores)

tprecisions = cross_val_score(classifier, x_test, y_test, cv=5, scoring='precision')
print('Test Precisions:',np.mean(tprecisions),tprecisions)
```

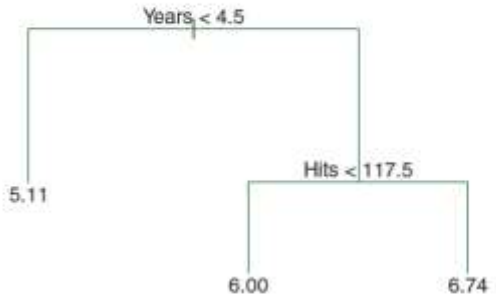
```
trecalls = cross_val_score(classifier, x_test, y_test, cv=5, scoring='recall')
print('Test Recalls:', np.mean(tprecisions),trecalls)

false_positive_rate,recall,thresholds = roc_curve ( y_test, predictions )
roc_auc = auc ( false_positive_rate, recall )
```

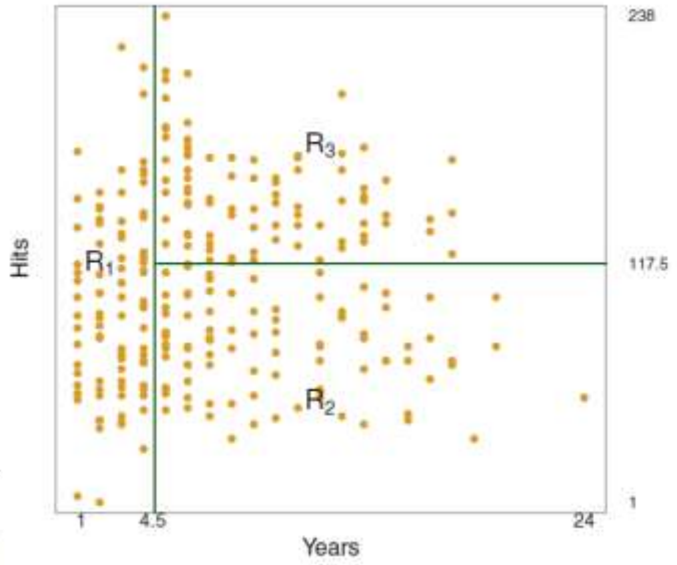
Decision Trees

What is Decision Tree ?

- Decision tree uses a tree structure to represent a number of possible decision paths and an outcome for each path
- Decision Trees can be applied to both classification & regression problems

Regression Tree for Predicting Log salary of Baseball Players





Classification
Criteria:
1] Entropy, Information Gain
2] Gini

Regression
Criteria:
1] Mean Square Error

$$R_1(j,s) = \{X|X_j < s\} \; R_2(j,s) = \{X|X_j \geq s\}$$

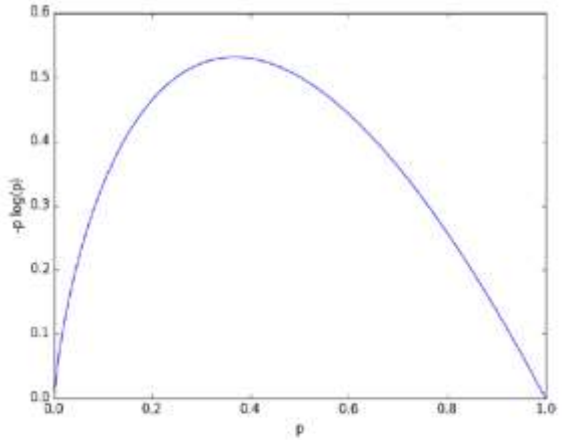
$$\sum_{i:\, x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:\, x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

Entropy = – p_1 * log p_1 …. – p_n * log p_n
p_i = proportion of data labeled as class C_i

Information Gain = Parent's Entropy – sum (weight of child * Child's Entropy)
Weight of child = (no.of observations in child)/total observations

Gini = 1 – $\sum p_i^2$
p_i = proportion of data labeled as class C_i

Entropy



Decision Trees – Grid Search

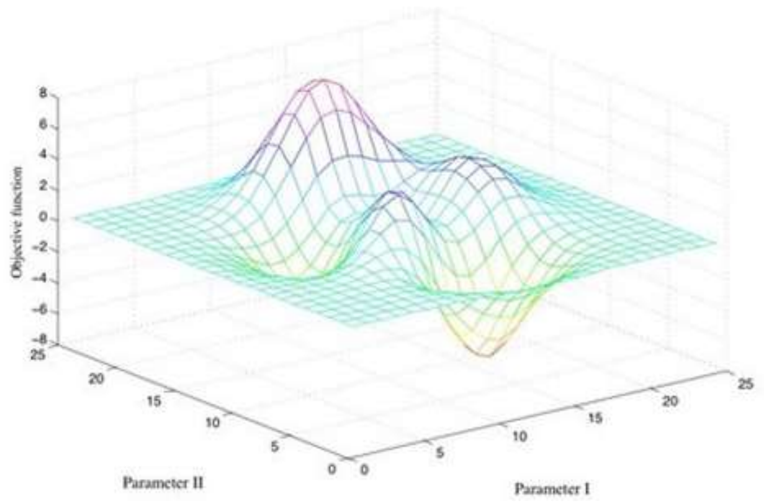
Grid Search – Classification Tree

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV

pipeline = Pipeline([
 (‘ clf ', DecisionTreeClassifier(criterion='entropy'))    ])

parameters = {
        'clf__max_depth': (150, 155, 160),
        'clf__min_samples_split': (1, 2, 3),
        'clf__min_samples_leaf': (1, 2, 3)
}

grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1, scoring='f1')

grid_search.fit (x_train, y_train)
predictions = grid_search.predict (x_test)

best_parameters = grid_search . best_estimator_ . get_params ( )

for param_name in sorted ( parameters.keys ( ) ):
    print ('\t%s: %r' % (param_name, best_parameters[param_name]))
```

Regression Tree

```
from sklearn.tree import DecisionTreeRegressor

pipeline = Pipeline([
 (‘ clf ', DecisionTreeClassifier(criterion=‘mse'))    ])
```

Grid Search



Random Forest

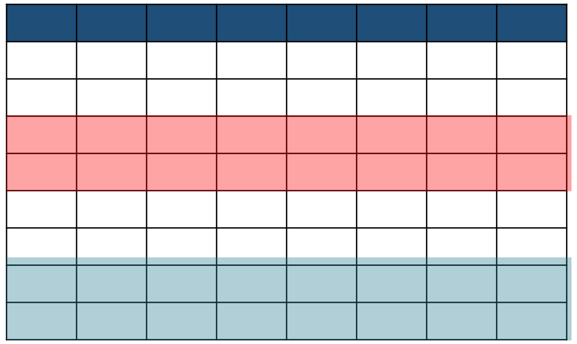
Random Forest

- Random forest is a collection of decision trees that have been trained on randomly selected subsets of the training instances and explanatory variables
- Random forests usually make predictions by returning the mode or mean of the predictions of their constituent trees
- Random forests are less prone to overfitting than decision trees because no single tree can learn from all of the instances and explanatory variables; no single tree can memorize all of the noise in the representation

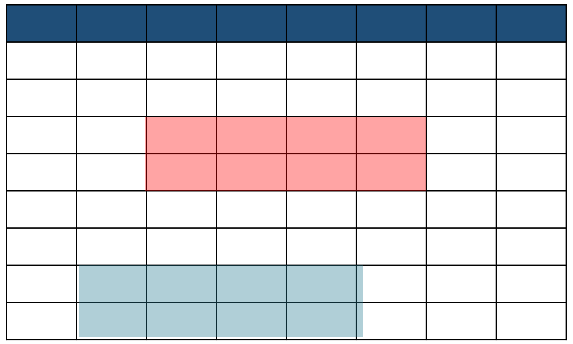
Each tree is trained on roughly 2/3 observations/training instances

No.of explanatory variables = p
Classification trees – Sqrt (p)
Regression trees – p / 3

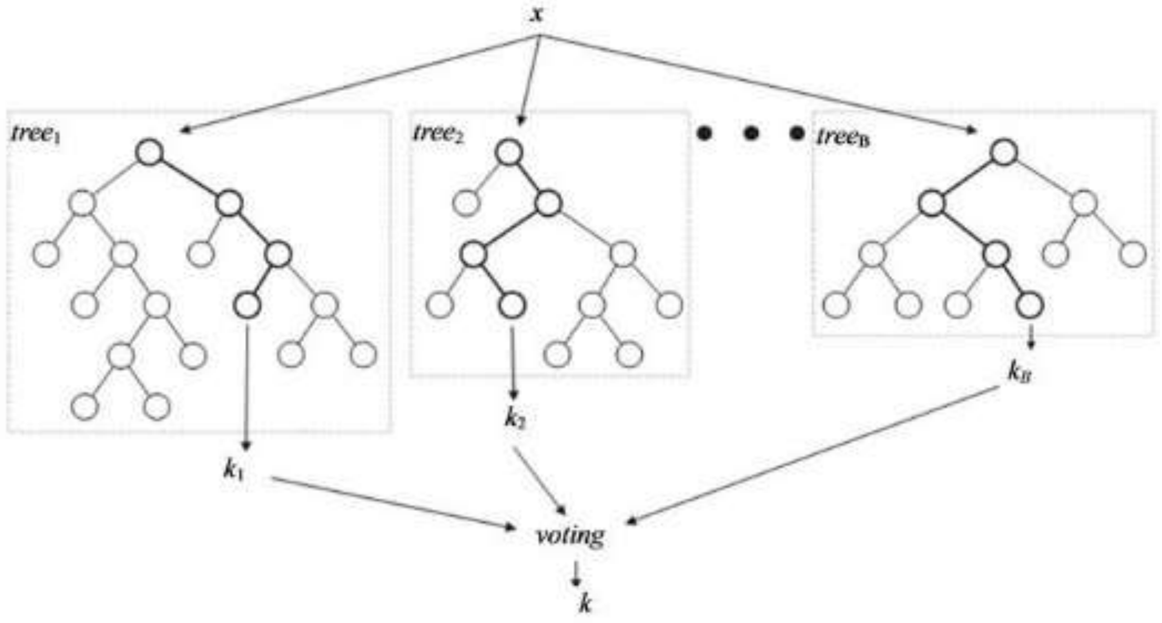
Bagging (Bootstrap Aggregation)



Random Forest



Random Forest Classifier

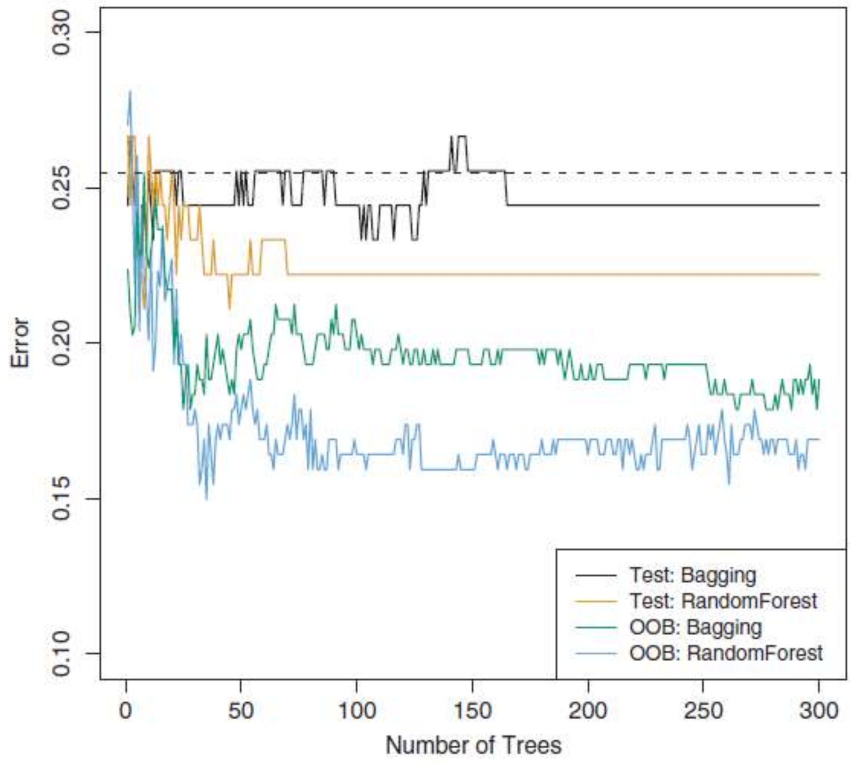


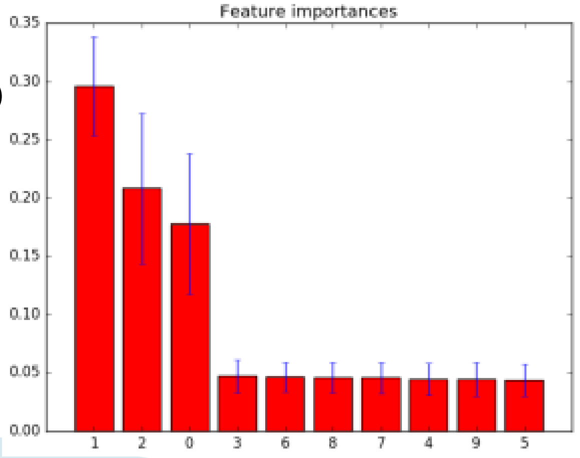
Random Forest

Grid Search – RandomForest Classifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV

pipeline = Pipeline([
 (‘ clf ', RandomForestClassifier(criterion='entropy‘, max_features=‘auto'))    ])

parameters = {
        'clf__n_estimators': (5, 10, 20 , 50),
        'clf__max_depth': (150, 155, 160),
        'clf__min_samples_split': (1, 2, 3),
        'clf__min_samples_leaf': (1, 2, 3)
}

grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1, scoring='f1')

grid_search.fit (x_train, y_train)
predictions = grid_search.predict (x_test)

best_parameters = grid_search . best_estimator_ . get_params ( )

for param_name in sorted ( parameters.keys ( ) ):
    print ('\t%s: %r' % (param_name, best_parameters[param_name]))
importances = grid_search.feature_importances_
```

Bagging vs. RandomForest





Boosting

Boosting

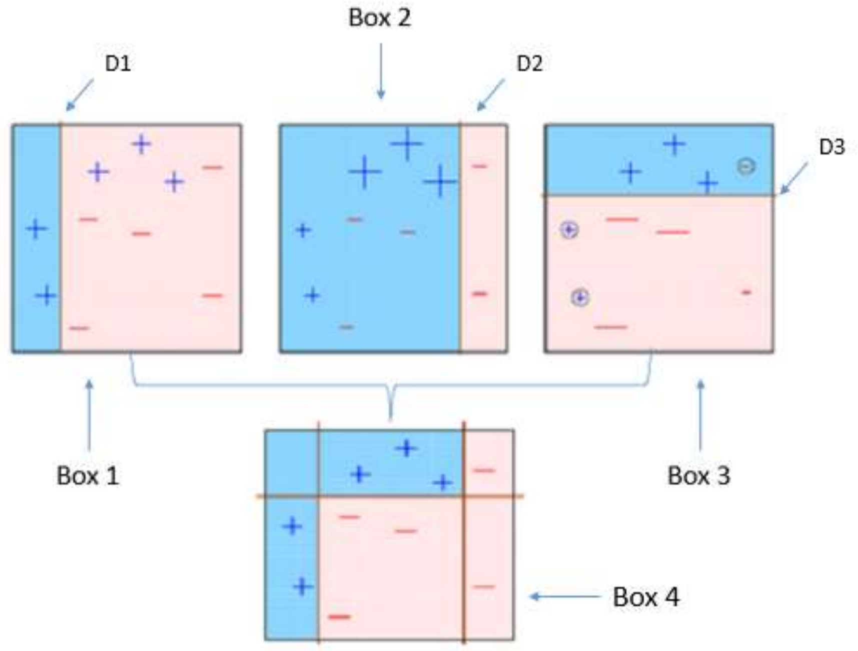
- Boosting refers to a family of algorithms which converts weak learner to strong learners

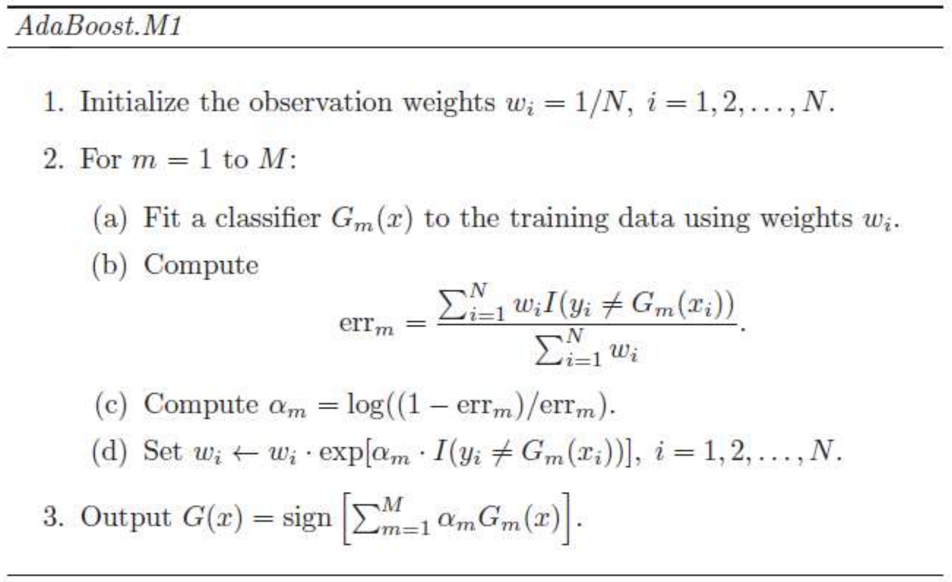
Steps in Boosting

- **Step 1:** Assign equal weights to all observations (E.g.: 1/N where N is No.of observations)
- **Step 2:** If there is any prediction error caused by first base learning algorithm, then we pay higher attention to observations having prediction error. Then, we apply the next base learning algorithm
- **Step 3:** Iterate Step 2 till the no.of models limit is reached or higher accuracy is reached

Finally, it combines the outputs from weak learner and creates a strong learner by taking a weighted mean of all boundaries discovered

Boosting Algorithm (Adaboost)

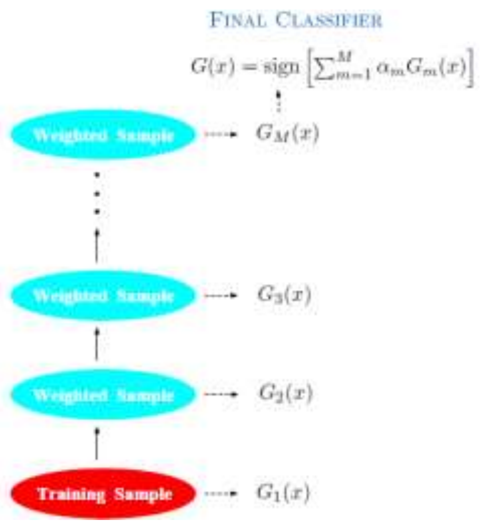




AdaBoost.M1

```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
            $\text{err}_m = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}$.
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^M \alpha_m G_m(x)\right]$.
```

🟨							
🟥							
🟨							
🟧							
🟨							
🟥							
🟧							
🟨							

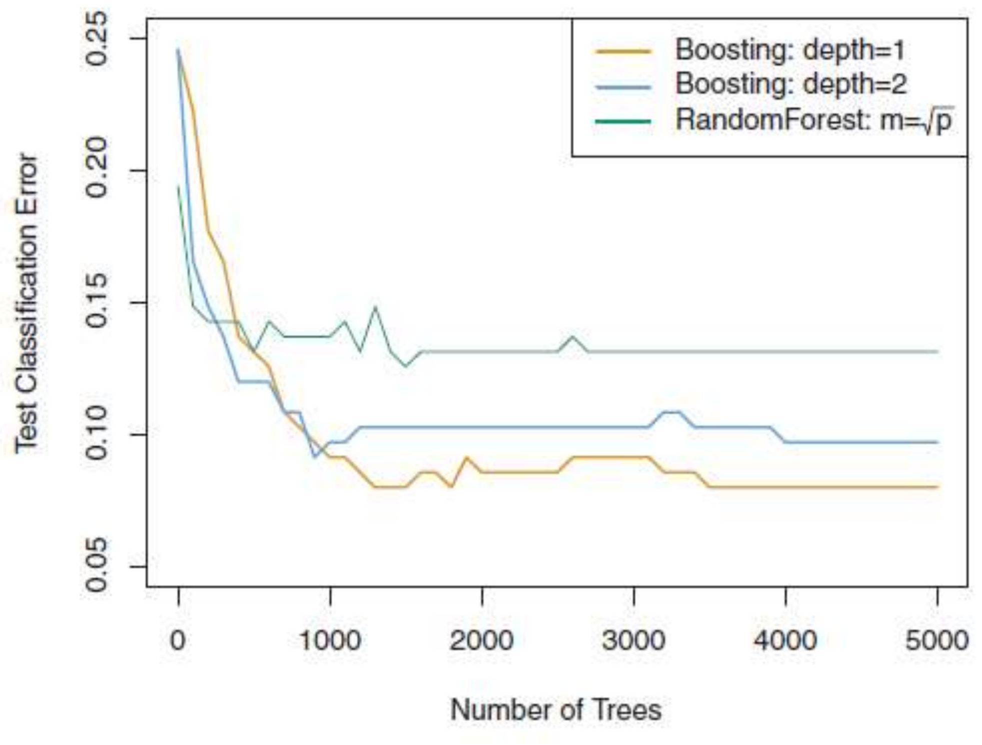


Boosting

Grid Search – AdaBoost Classifier

```
from sklearn.ensemble import AdaBoostClassifier


from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV

pipeline = Pipeline([
 (' clf ', AdaBoostClassifier( base_estimator = DecisionTreeClassifier(max_depth=1),
 algorithm = "SAMME.R" ))
   ])

parameters = {
       'clf__n_estimators': (50, 100, 200 , 500),
       'clf__learning_rate': (0.1, 0.5, 1),
}

grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1,
scoring='accuracy')

grid_search.fit (x_train, y_train)
predictions = grid_search.predict (x_test)

best_parameters = grid_search . best_estimator_ . get_params ( )
```

Boosting vs. Random Forest

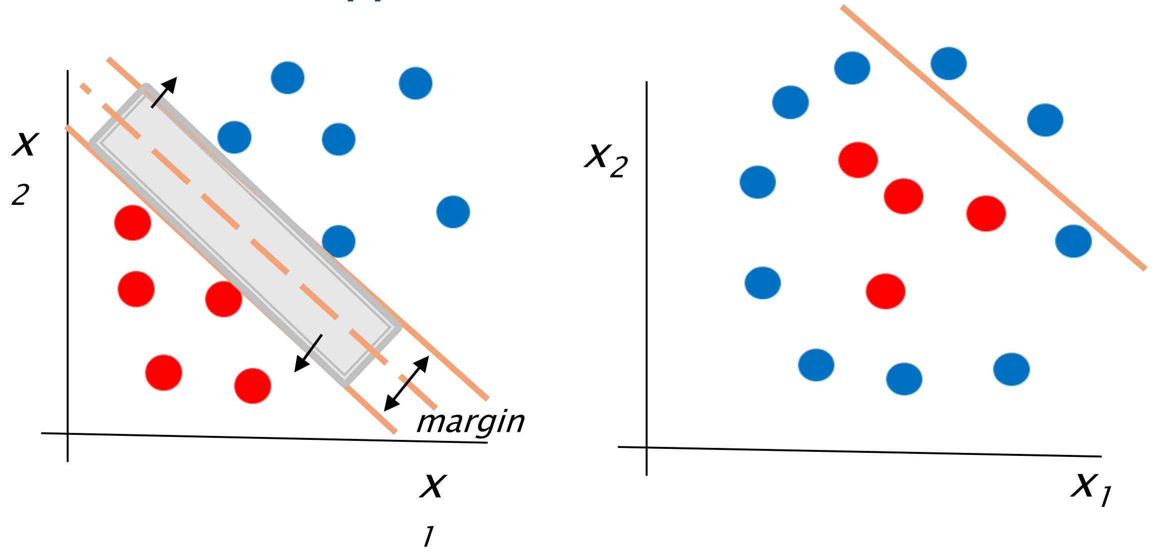


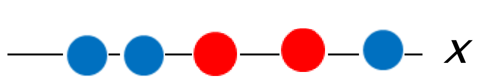
Support Vector Machines

Support Vector Machines

- Support Vector Machine maximizes the margin between different classes
- When the data is not linearly separable, SVMs use the "Kernel Trick" to map data to higher dimensions using Kernel Matrices
- A Kernel Matrix is the inner product of the mapping of the data points

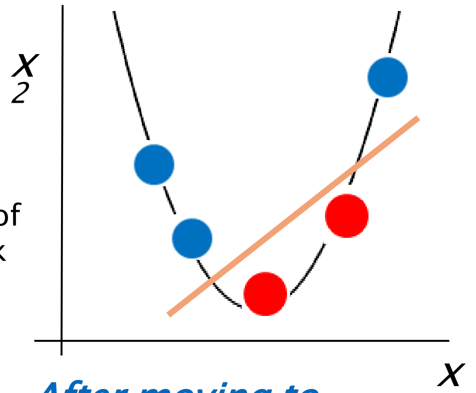
Support Vector Machines





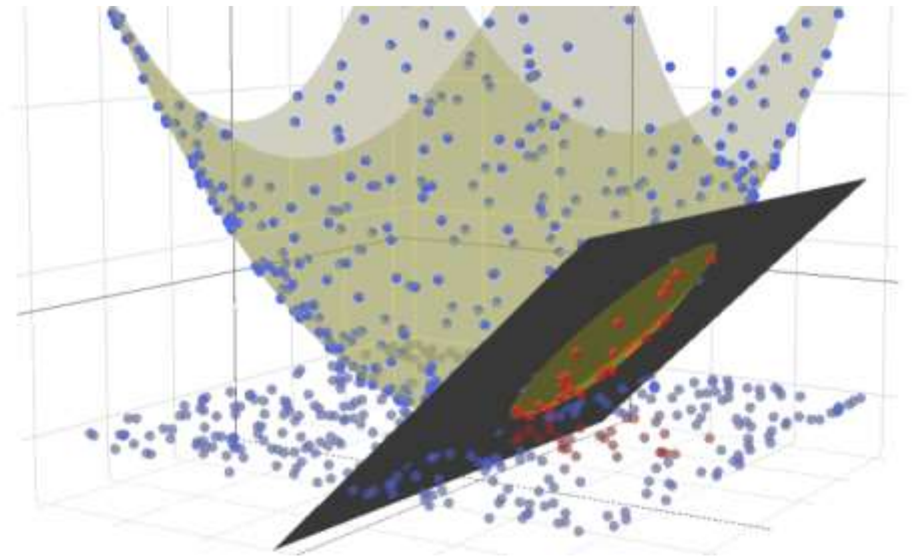
1D example of Linearly Inseparable Case

Application of Kernel Trick on 1D data



After moving to higher dimensions, data will be Linearly Separable

Kernel trick on 2D example

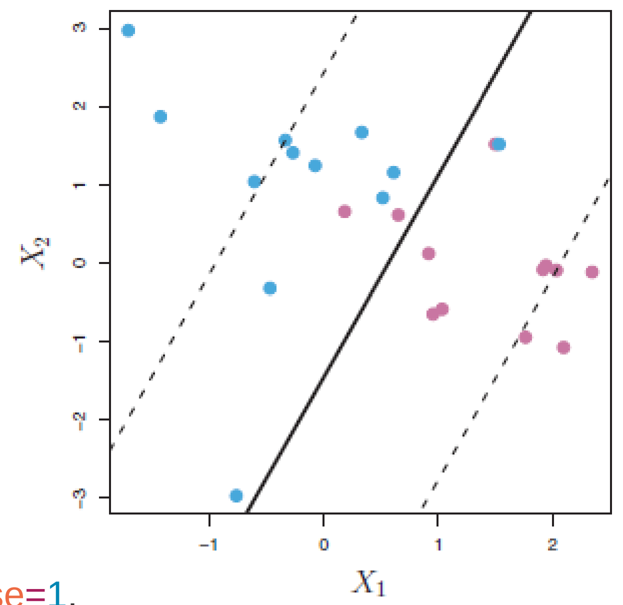


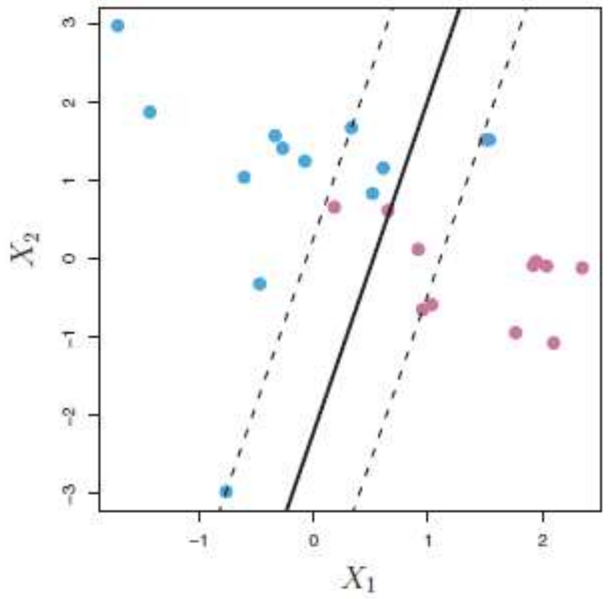
Support Vector Machines

Support Vector Classifier

```
from sklearn.svm import SVC

from sklearn.pipeline import Pipeline
from sklearn.grid_search import GridSearchCV

pipeline = Pipeline([
 (‘ clf ', SVC ( kernel =  ’rbf‘))
    ])

parameters = {
        'clf__gamma': (0.001, 0.03, 0.1 , 0.3 , 1 ),
        'clf__C': ( 0.1, 0.3, 1 , 3, 10 , 30 )
}

grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1,
scoring='accuracy')

grid_search.fit (x_train, y_train)
predictions = grid_search.predict (x_test)

best_parameters = grid_search . best_estimator_ . get_params ( )
```

Impact of Cost on Margins





Margins will be closer as cost of violations C decreases i.e. variance increases with the decrease in cost

$$\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n}{\text{maximize}} \quad M$$

$$\text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M(1 - \epsilon_i),$$

$$\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C,$$

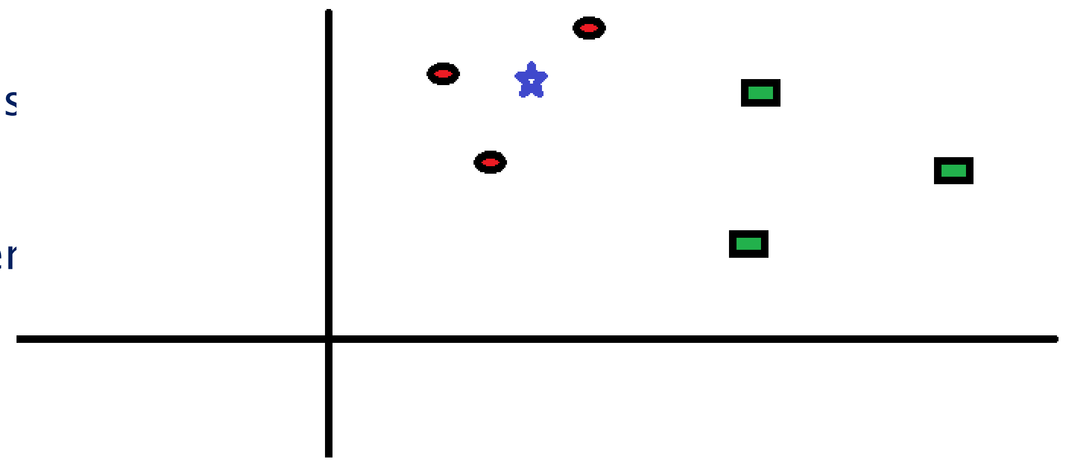
K Nearest Neighbors

K Nearest Neighbors

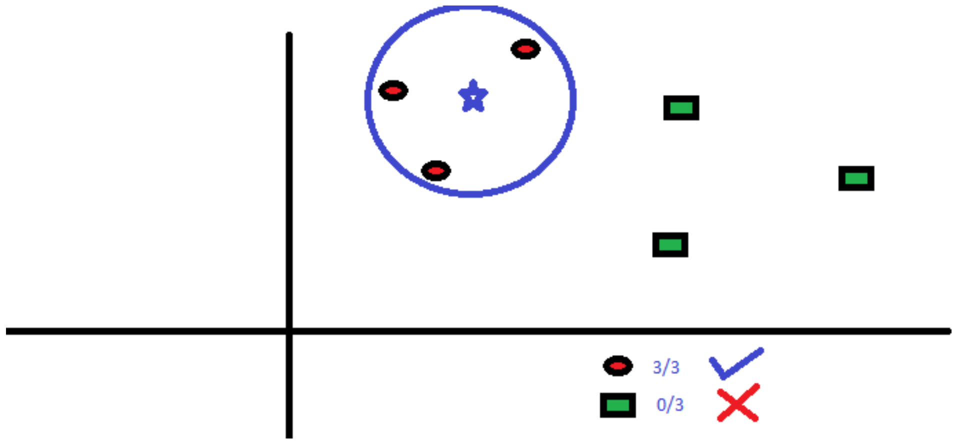
- K Nearest neighbors is one of the simplest predictive models. It makes no mathematical assumptions, and it doesn't require any sort of heavy machinery. The only things it requires are:
  - Some notion of distance
  - An assumption that points that are close to one another are similar

```
from sklearn import neighbors
knn = neighbors.KNeighborsClassifier ( n_neighbors = 3)
knn.fit (x_train, y_train)
predicted = knn.predict (x_test)
```

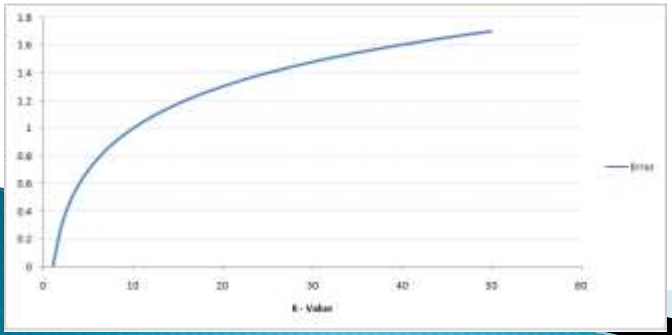
K Nearest Neighbors



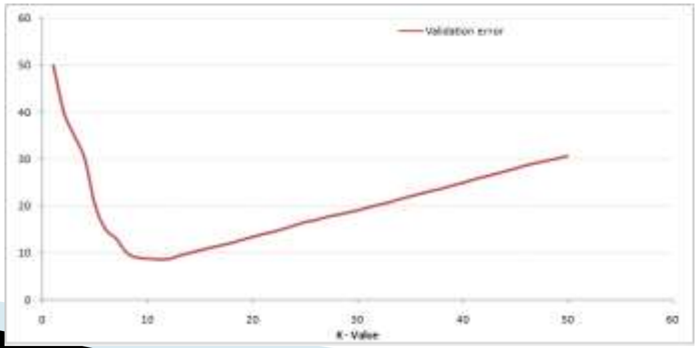
K Nearest Neighbors (K=3)



Training Error



Validation Error



Neural Networks

Neural Networks

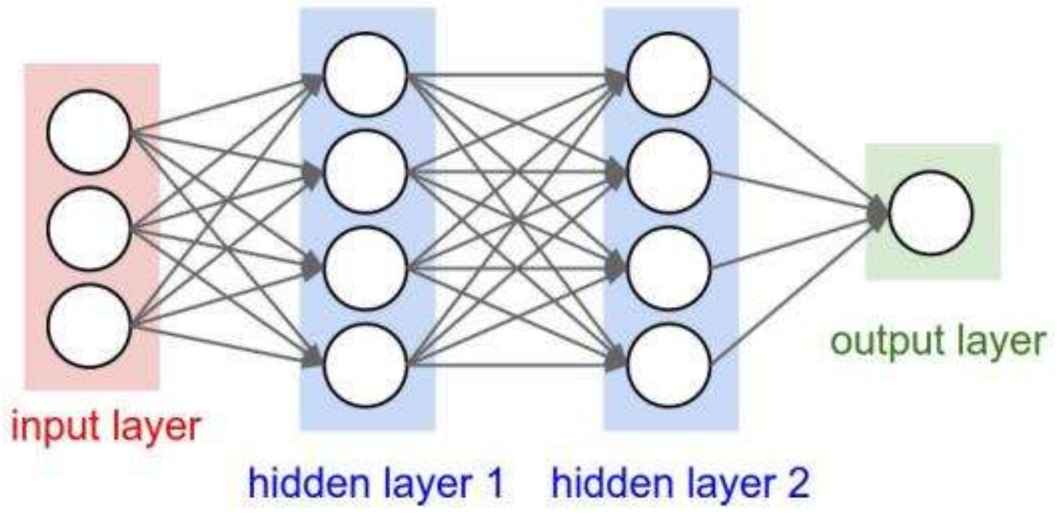
- Neural networks are analogous to human brain

```
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler

clf = MLPClassifier( activation ='relu', solver='lbfgs', alpha =1e-5,
hidden_layer_sizes =(5, 2), max_iter = 200, random_state =1)

scaler = StandardScaler ( )
scaler.fit (x_train)
x_train = scaler.transform (x_train)
x_test = scaler.transform (x_test)

clf.fit (x_train, y_train)
clf.predict (x_test)

print [coef.shape for coef in clf.coefs_ ]
[(2, 5), (5, 2), (2, 1)]
```

Sample x, y

```
x = [[0., 0.], [1., 1.]]
y = [[0, 1], [1, 1]]
```

Two hidden Layer MLP (Multi Layer Perceptron) Neural Network



Naïve Bayes

Bayes Theorem

Prior Probability

Likelihood of the evidence 'E' if the Hypothesis 'H' is true

$$P(H|E) = \frac{P(H) * P(E|H)}{P(E)}$$

Posterior Probability of 'H' given the evidence

Priori probability that the evidence itself is true

Naïve Bayes

- Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features

```
from sklearn.naive_bayes import GaussianNB

clf = GaussianNB ( )
clf.fit ( x_train,  y_train )
predict = clf.predict (x_test)
```

What is the probability of email is spam when following words appear ?
Lottery = yes, Money = no, Groceries = no, Unsubscribe = yes

$$P(\text{Spam} \mid W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) = \frac{P(W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4 \mid \text{spam})\, P(\text{spam})}{P(W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4)}$$

Intersection operations on words are expensive, a naïve independence assumption improves Computational efficiency, yet effective in providing results

$$P(\text{Spam} \mid W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) = \frac{P(W_1 \mid \text{spam})\, P(\neg W_2 \mid \text{spam})\, P(\neg W_3 \mid \text{spam})\, P(W_4 \mid \text{spam})\, P(\text{spam})}{P(W_1)\, P(\neg W_2)\, P(\neg W_3)\, P(W_4)}$$

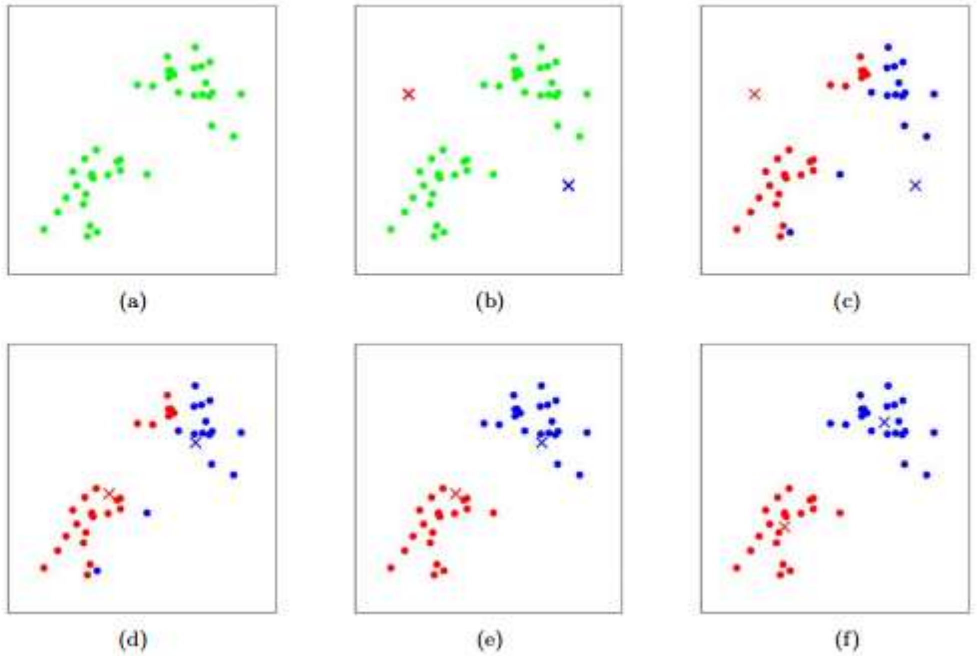
K means Clustering

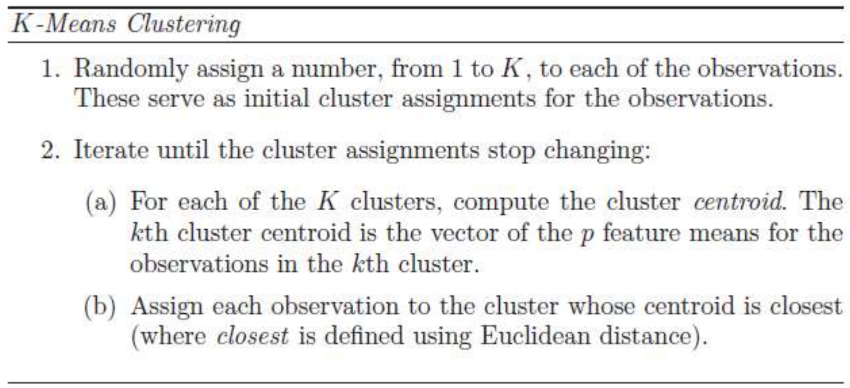
K means clustering

- K–Means is an iterative process of moving the centers of the clusters, or the centroids, to the mean position of their constituent points, and re–assigning instances to their closest clusters

```
from sklearn.cluster import Kmeans
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
K = range(1,10)
meandistortions = []
for k in K:
    kmeans = Kmeans (n_clusters=k)
    kmeans.fit (x)
    meandistortions.append (sum( np.min (cdist
(x, kmeans.cluster_centers_, ' euclidean ') ,axis=1)) / x.shape [0] )
plt.plot (K,meandistortions,'bx-')
```

K Means Clustering

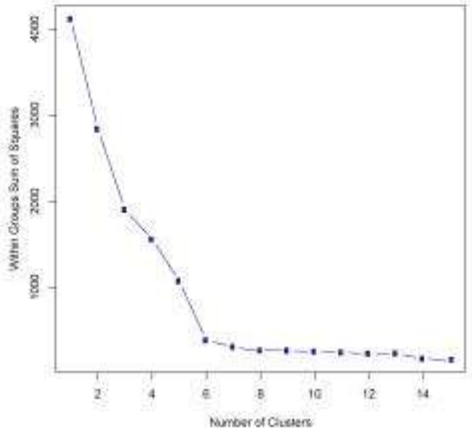




K-Means Clustering

1. Randomly assign a number, from 1 to K, to each of the observations. These serve as initial cluster assignments for the observations.
2. Iterate until the cluster assignments stop changing:
   (a) For each of the K clusters, compute the cluster *centroid*. The kth cluster centroid is the vector of the p feature means for the observations in the kth cluster.
   (b) Assign each observation to the cluster whose centroid is closest (where *closest* is defined using Euclidean distance).

Elbow plot



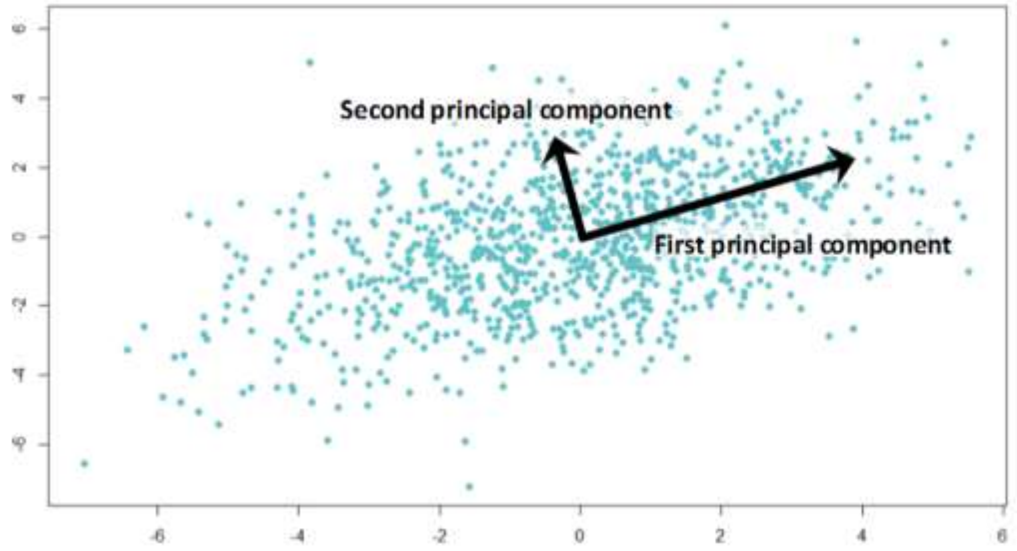
Principal Component Analysis

Principal Component Analysis

- PCA reduces the dimensions of a data set by projecting the data onto a lower-dimensional subspace
- PCA reduces a set of possibly-correlated, high-dimensional variables to a lower-dimensional set of linearly uncorrelated synthetic variables called principal components
- lower-dimensional data will preserve as much of the variance of the original data as possible

Various dimensions of Watering Can

PCA with orthogonal rotation on 2D Data

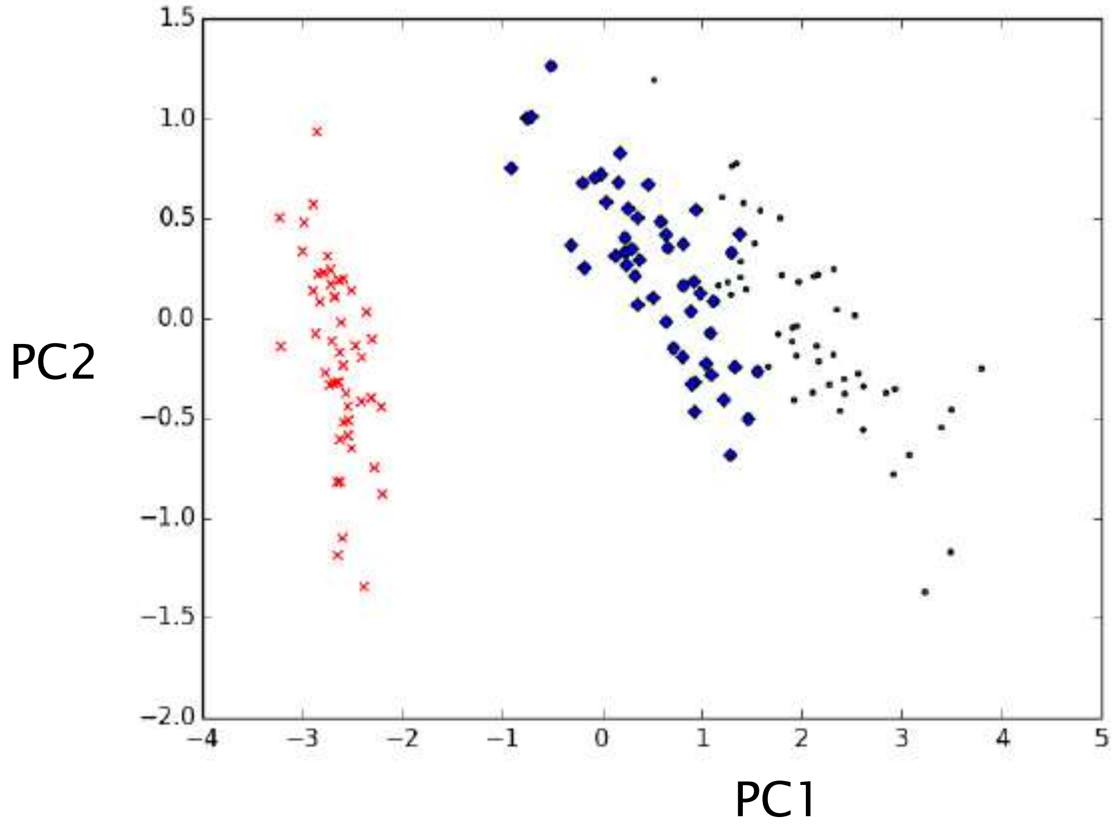


Watering Can seen from Principal Component (Direction of Maximum Variance)

Principal Component Analysis

```
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris

data = load_iris( )
y = data.target
x = data.data

pca = PCA ( n_components = 2 )
reduced_x = pca.fit_transform ( x )
```

2D representation of original 4D IRIS data



Thank You