Decision Tree

Recommended Use for:

1. Discrete Class values
2. Noise in training Data

Algorithm:

Input: Set of training data D

Output: Hypothesis H in the form of a decision tree

* Create Root node with all instances in D
* If node is impure(i.e not all instances classified to the same class value)
  + Find best attribute xi to split instances
  + For every value v of attribute xi
    - Create new branch and add to its child node every x in D such that xi=v
* Repeat until all nodes are pure lol

Choosing best Attribute

* Purity : A set of instances S is considered pure if all of its instances are classified to the same value.
* Impurity: Impurity is when you have at least two instances within the set with different class values.

*How to measure impurity:*

* + Given a set of instances S, with class attribute C that has k different possible values, we will define a group P={P1,P2,..,PK) where Pi is the probability of choosing an instance with class value i from S
  + The impurity of S is defined by a function (P) where:
  + : [0,1]^k→ R
  + (P)>=0
  + (P) is minimal if S is pure
  + (P) is maximal if all probabilities are distributed equally
  + (P) is symmetric
    - For example: P1={0.3,0.7}, P2={0.7,0.3} → (P1)=(P2)

Goodness of split

The goodness of split of S with respect to attribute A is the difference between the Impurity of S before split (defined by (S)) and the overall impurity after splitting:

\Delta \phi (S,A) = \phi (S) - \sum\limits_{v \in A} {\frac{{|{S_v}|}}{{|S|}}*\phi ({S_v})} 

Therefore, when choosing the best attribute, we will choose the attribute with the highest goodness of split

Examples of Impurity Functions

* 1. Function: Gini Index

GiniIndex(S) = 1 - \sum\limits_{} {{{({p_i})}^2}} 

* 1. Goodness of Split: Gini Gain



* 1. Function: Entropy



* 1. Goodness of Split: Information gain



Overfitting and fixing it

Overfitting - When our hypothesis is too specific, training data has low error but for newly added data the error is large.

Given training data S, data D (S \subset D) , Hypothesis H

We will say that H overfits S if there exists another hypothesis H’ such that:

errortrain(H)<errortrain(H’) and errorD(H) > errorD(H’)

Fixing

There are two main methods to fix overfitting of our hypothesis:

1. Stop growing before tree is full
   1. Use a validation test with your training set,
   2. When building the tree using the training set, test the error for validation set with each branching
   3. Once the error has increased from previous branching, stop growing the motherfuckin tree
2. Pruning nodes after tree is full
   1. Build the tree using training set
   2. Calculate error of validation set
   3. For every node calculate the new error of the validation set without the node (Chi square, convert tree to rules).
   4. Remove the node that improves the error of the validation set the most

Perceptron

Input: Training data D and learning rate 

Output: Hypothesis h that classifies instances to + or -

Algorithm

1. Set weights to W1...Wn to initial random values
2. For each 
   1. Calculate: o({x_d}) = sign(\vec w \cdot \vec x) = sign({w_0} + {w_1}{x_1} + {w_2}{x_2} + ... + {w_n}{x_n})
3. For each 
   1. Calculate \Delta {w_i} =  - \eta  \cdot {x_i} \cdot \sum\limits_{d \in D} {(o({x_d}) - t({x_d}))} 
   2. Update :



4. Repeat from 2

* If training data is linearly separable, and  is small enough, perception is guaranteed to achieve 0 error on training data (consistent learner)

Gradient Descent

Definitions:

Cost Function- The sum of distances of all instances values from our YORO function



Where:

- instance i

- class value of instance i



m - number of training instances

- weights vector

- learning rate

Goal of gradient descent:

Find weights vector () that minimizes the cost function J

Algorithm:

1. Set weights to random values
2. For every {\theta _j}

{\theta _j} = {\theta _j} - \alpha  \cdot \frac{\partial }{{\partial {\theta _j}}}J(\vec \theta )

1. Repeat until minimum reached

Partial derivative of J()”



KNN- K Nearest Neighbors

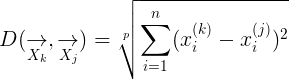
Parzen Window - Given training data D, in order to classify new instance x, create a window of size h around x. Look at all instances in D that fall into this window, and classify x to the majority class value of these instances.

Problem: There may not be any instances within the window, then we wouldn’t know how to classify

Solution: choose how many instances you want in your window, and set window size accordingly (KNN)

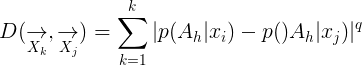
Finding nearest neighbors

Given instance x, need to find k nearest instances within the training data. We have different techniques to measure distance between two instances:

1. LP Distance: 
   1. If p=2 Euclidean distance
   2. If p=1 manhattan distance
   3. If p= infinity then :



2. Value difference measures



Note: Usually, q=1,2

How to improve efficiency of KNN

It takes O(m) to run KNN, where m is the number of instances

We can improve the efficiency of KNN using several methods:

1. Reduce distance calculation time:
   1. Calculate Lp Distance for r<n where n is the number of dimensions. r should limit the size of the sum to a threshold.
2. Reduce search time using search structures
   1. Use KDTree: given training data:
      1. Cycling through all dimensions of instance x:
         1. Find the median value by current dimension
         2. Sort instances to groups with smaller and higher values of median value in current dimension
         3. Recursively continue to both groups with next dimension

This defines regions in the instance space, and when given a new instance, the instances in its region will be the nearest neighbors.

3. Select only attributes that affect the boundary:

1. Forward Wrapper:

Insert instances to training data one by one, keeping only those that are NOT classified correctly.

b. Backwards Wrapper:

Start with all instances and remove those that are classified correctly.

X-Fold cross validations

* Used when there is not enough data for training and validation set
* Gives an estimation to the true error based on training data

Algorithm

1. Split training set into X parts
2. Use all but 1 as training data and build classifier
3. Test error on remaining part
4. Repeat for every part
5. Average all errors

KNN-Summary

Pros:

* Fast training
* Learn complex target functions
* Does not lose information

Cons:

* Slow query time
* Takes a lot of space
* Can be fooled by irrelevant attributes

Support Vector machine

Problem:

Given data is not linearly separable.

Solution:

* Map data to higher dimensions in hope that data is linearly separable there.
* So for example we can create a mapping function, lets call it Shdema.
  + 
  + We can use our Shdema function inside the perceptron algorithm, thus solving the problem we mentioned earlier:
    - For each :
      * 
    - For each :
      * 
  + Using a Shdema function can be extremely inefficient and costly because working in higher dimensions require us to process much more information
  + Also a shdema function has few limitations
    - We can only reach a finite dimension
    - We are limited by computation power

Overcoming working in higher dimension:

* Introducing Kernel trick, a trick that Arik Shamir uses in bed.

Kernel Trick:

Let's call your transformation function F. Most linear machine learning techniques can be implemented with using only the dot product operation, call it P. If you can compute a . b for any two points of your dataset, you often don't need anything else - you don't need to even know the dimensionality of the points.

So what if you knew a function K(a, b) such that K(a, b) = F(a) . F(b). Then, during learning, every time you needed to compute F(a) . F(b), you'd just compute K(a, b) - you wouldn't need to map to higher dimension thus increasing efficiency greatly.

Let's go back to the perceptron Algorithm:

We can show that the final weight vector  is equivalent to:



- reflects the “difficulty” of classifying  (i.e the more we misclassify , the bigger  will be). More on  later

 - target class value

- instance vector

Now back in original perceptron:



We substitute  with the above summation:



Now, to map to higher dimension we’ll use Shdema function:



And finally in order to be efficient, we will use a kernel instead of a mapping function:



Notice that only instances where , (Instances that were misclassified) are relevant to the summation

These instances are called support vectors

Instead of updating  like in the original perceptron algorithm:



We can update :

If 



Finally the new kernel perceptron algorithm will look like that:

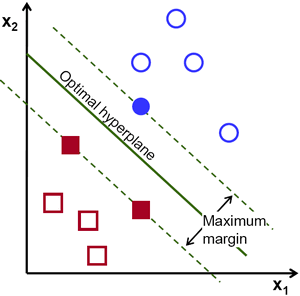
1. Initialize feature weights  randomly

This will give us a separating hyperplane to our data

Finding the best hyperplane

Using kernel perceptron algorithm we can find a hyperplane that can separate our data, but is it the best? Can we improve it?

Consider the following illustration of hyperplane and some data:



Margin- The distance between the separating hyperplane to the instance closest to it

Our goal- maximize the margin in order to get optimal hyperplane

The absolute distance between an instance  to the separating hyperplane is:



Let's denote the margin as b, so for every instance :



Our goal is to maximize the distance of each instance, therefore our goal is to maximize b

We will define: 

And we will try to minimize w in subject to the constraint defined above:



Substitute b:



Which equivalent to:



Note: Minimizing  are equivalent operations

Lagrange multipliers methods

* A strategy for finding local max and min values of a function subject to certain constraints
  + So to maximize f(x,y) subject to g(x,y)=0
  + Define a lagrangian multiplier 
  + The lagrange function will be:
  + 

Using partial derivatives of L(x,y) and the constraints, we can build a system of linear equations that will find critical points (max, min) for x,y

Then plugging these x,y values into f(x,y) will find max, and min points

Minimizing  with constraints to  for every 



- Our lagrange multipliers

What if data is still not linearly separable in mapped dimension

* We can allow a certain number of instances to be classified incorrectly
* We measure this with a “slack variable”
* S our new goal is to minimize  subject to : 
* 

-We can update our goal part of our Lagrange function to minimize our slack:

* 
* So our new 

Reminder for Lagrange:

* + 



- Lagrange multiplier

- instance i

-Feature Weight vector

-target class value of instance i

- distance of the hyperplane to the origin

-the distance between hyperplane to the misclassified instance

K- control the trade off between the slack variable

Learning theory

- the error of hypothesis h on the training data.

- the error of hypothesis h on all data

We want to find a close estimation of  using only our training data and 

Since we don’t know the true error, we will use a generalization error:

Generalization error:

A measure of how accurately an algorithm can predict values for previously unforeseen data.

Examples:

* The error on the training data
* K-fold cross validation

How do we find the algorithm that will give best true error?

Generalization accuracy of learner L is the accuracy of L on all non-training instances

Denoted :

For any L, the average GA over all concepts in F is 0.5:



*bias* is the amount by which the expected hypothesis prediction differs from the true value or target

* Bias is the true error of the best classifier in the concept class (e.g, best linear separator, best decision tree on a fixed number of nodes).
* Bias is high if the concept class cannot model the true data distribution well, and does not depend on training set size.

*variance* is the amount by which the hypothesis will change if one instance were to change its class value.

* Variance is the error of the trained classifier with respect to the best classifier in the concept class.
* Variance depends on the training set size. It decreases with more training data, and increases with more complicated classifiers

Probably approximately correct (PAC) framework

Goal: prove that a learner L can probably output a hypothesis h that achieves a certain true error.

For example:

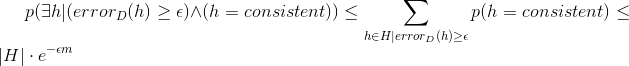
“There is 95% chance that I can find a classifier that will have at most 5% error on the data”

A hypothesis h is called  bad if 

The probability that a random instance will be misclassified is 

The probability that m random instances will be classified correctly is 

Which is also the probability that h is consistent with m samples



If we bound this probability to a certain lambda:



We can extract m and get:



This essentially bounds the minimum number of instances needed in our training data

Example:

Assume we have 10 boolean attributes

Using conjunctions as our hyper space, we get 

We want to ensure with 95% certainty that h will have error<10 %





So we need at least 140 instances

PAC Learnability

C- possible group of concepts

L- learning algorithm

H - hyperspace

C is PAC learnable by L using H, if for every , learner will output a hypothesis h in the probability of 1-, such that  in time that is polynomial. The time should be expressed only using:



Checking for PAC learnability

Conditions:

1. |H| is finite
2. Concept is in hypothesis space

Steps to check:

1. Define a consistent learner
2. Check if sample complexity is polynomial using : 
3. Check if each step in the learner is polynomial

If concept is not in hyper space we will use chernoff-bounds for instance bounds:



If hyper-space if infinite:

Use VC-Dimension to measure complexity of hypothesis

Shattering

A set of instances S is **shartted**, by hypothesis space H,if for every dichotomy of S, there exists a hypothesis in H which is consistent with the dichotomy

VC Dimension

VC(H) is the size of the largest finite subset of instance spaceX that is shatterable by H

Feature Extraction & selection

* Having problems in high dimensions can help us classify better
* However adding too many features can hurt our classifier
  + Irrelevant features that create wrong patterns
  + Feature space grows exponentially
    - Instances become sparse
    - Hard to generalize
* Our goal: Find a smaller set of features that represent our data better
* There are 2 methods to find such features
  + Feature Selection
    - Find subset of features that are the most relevant to the problem and use them for classification
    - Examples:
      * Wrapper methods: select the best features that optimize machine learning performance
        + Algorithm for forward wrapper:
        + Start with empty set of features
        + Add feature that increases the classifier’s accuracy the most
        + Repeat until predefined accuracy (threshold) or number of features achieved
      * Filter methods: choose features that are correlated with class value (Pearson correlation)
  + Feature extraction
    - Find new set of features that better represent the data (lower the dimension of our instance)
    - Methods:
      * PCA:
        + Main idea: Transform attribute’s vector space to a new basis that better describe the data
        + Objective: transform the coordinate system by moving and rotating it