Recap

- We have looked at some generic issues in learning from examples.
- We looked at No Free Lunch theorem to say that there is no universally best algorithm to learn from examples.
- We also looked at the concept of bias and variance of a learnt model.

Recap

- We have discussed some practical methods for model selection and assessment.
- For the model learnt with training data, the error on training data is not a good estimate of its true (generalization) error.
- We can use hold out validation set to assess the learnt model.
- Since the validation set is independent of the learnt model, the empirical risk on validation set is a good estimate of true risk. (E.g., we can bound the estimate using Heoffding's inequality).

Recap

- We can use a validation set for model selection or hyperparameter tuning also.
- We repeatedly learn different models using the training set and choose among them using validation set.
- The validation part is like empirical risk minimization over a finite hypothesis space.

Recap

- If we are trying to learn a complex model, we may "overfit".
- Overfitting is indicated by having small error on training data but large error on validation data.
- This can be studied using the model selection curve.
- We should try and control the complexity of the hypothesis space (for the number of examples we have)

Recap

- We also need a good estimate of the final learnt model (with the hyperparameters selected through validation).
- We use a separate test set for that.
- The data is divided into Training, Validation and Test sets.
- A thumb rule is to use 50% for training and 25% each for validation and test.
- The split is to be done randomly.

Recap

- When data is scarce, we cannot afford to use only 50% of data for training.
- In such cases one uses K-fold cross validation
- We divide data into K parts and repeat training algorithm K times.
- Each time we use K-1 parts for training and one part for validation (or estimating error)
- The final error estimate is average of all errors.
- Cross validation can also be used for model selection and tuning hyperparameters
- One normally uses 5 or 10 fold cross validation.
- The extreme case is leave-one-out cross validation.

- We are using validation to get an estimate of the true risk.
- We are learning $\hat{h}^*$, a minimizer of the empirical risk over training set, $\hat{R}_{tr}$. (We are not showing the number of training samples to keep notation simple).
- The generalization error of the learnt model is $R(\hat{h}^*)$.
- Let $\hat{R}_v$ be the empirical risk on validation set.
- Since validation set is not used in learning $\hat{h}^*$, $\hat{R}_v(\hat{h}^*)$ is a good estimate of $R(\hat{h}^*)$.
- This is the idea in hold out validation or K-fold cross validation.

- We can write

$$R(\hat{h}^*) = \Big(R(\hat{h}^*) - \hat{R}_v(\hat{h}^*)\Big) + \Big(\hat{R}_v(\hat{h}^*) - \hat{R}_{tr}(\hat{h}^*)\Big) + \hat{R}_{tr}(\hat{h}^*)$$

- The first term can be tightly bounded using Hoeffding inequality (if we have enough validation examples).
- The last term is the training error. Suppose it is small.
- Then only the middle term contributes to high true risk of the learnt model – overfitting.
- When there is overfitting we may want to increase the number of training samples or reduce the hypothesis space.
  (Note that when VC dimension is finite, both the training and validation errors converge to the minimum of true risk as number of examples go to infinity).

- When we have small training error and large validation error, we called it overfitting.
- But increasing the training set size may not always solve the problem.
- Suppose $R(h^*)$ is large.
- Then validation error would always be large because it is, at best, a good approximation of $R(\hat{h}^*)$ which is larger than $R(h^*)$.
- Note that $R(h^*)$ depends on the hypothesis space and underlying data distribution, and not on the learning algorithm or training data.
- In such a case we may have to change the hypothesis space (model class) or the input.

- When we have small training error and large validation error, we may first want to look at the learning curve.
- We can plot training and validation errors versus training set size (by training on different subsets of training set).
- If we see some trend of decreasing validation error with increasing training set size, then we can hope to beat overfitting by getting more training data.
- If the training data is expensive we may have to try and reduce complexity of hypothesis space.
- If there is no decreasing trend in validation error, then the problem may be with $R(h^*)$.
- Then it may be better to first try and change the hypothesis space or the input representation.

- As opposed to this, suppose the training error remains large.
- Then it is referred to as underfitting.
- This is most probably due to large $R(h^*)$. (Assuming we are learning empirical risk minimizer):

  $$\hat{R}_{tr}(\hat{h}^*) = \left(\hat{R}_{tr}(\hat{h}^*) - \hat{R}_{tr}(h^*)\right) + \left(\hat{R}_{tr}(h^*) - R(h^*)\right) + R(h^*)$$

  Since the first term is negative and second one can be bounded well and does not depend on learning algorithm, large training error is due to large $R(h^*)$.
- So, we want to expand the hypothesis space or change hypothesis space or change input representation.
- Here, just increasing examples may not help.

- We employ cross validation when we do not have sufficient data to make training/validate/test split.
- In cross validation, we are essentially generating multiple (overlapping) training sets and using the same model class and algorithm to learn model on each of these.
- This averaging is what may be giving us a reasonable estimate of the true error of model we learn.
- There are other methods to generate such multiple training sets for estimating generalization error.

Bootstrap Methods

- Bootstrap estimates provide another general method for estimating the generalization error.
- The idea in bootstrap is to generate many training sets by sampling with replacement from the given data.
- Given the original data of n points, we generate B number of training sets, each of size n, by randomly sampling from the given data set.
- Then we learn a model on each of the B training sets.
- The final error estimate could be the average of errors of all the models.

- Unlike in cross-validation, here we can have as many training sets as we want (all with size n).
- However, they may all be very similar.
- Let $\hat{f}^b$ denote the model learnt using the b^{th} bootstrap sample, $b = 1, \cdots, B$.
- The final bootstrap estimate of error is

$$e^1_{boot} = \frac{1}{B} \sum_{b=1}^{B} \frac{1}{n} \sum_{i=1}^{n} L(\hat{f}^b(X_i), y_i)$$

- Here we are using the original data set as test data while for each b, the $\hat{f}^b$ is learnt using some of the same data.
- Hence this bootstrap error estimate would not be very good.

- As an example, consider a problem where the class label is independent of the feature vector.
- Then the true error rate is 0.5 (under 0–1 loss).
- Suppose we use the 1-nearest neighbour as our classifier.
- Then on any X_i, the classification by $\hat{f}^b$ would be correct if X_i is in the b^{th} bootstrap sample.
- Otherwise, it would be correct with probability 0.5.
- Now,

$$\begin{aligned} P[X_i \in \text{bootstrap sample b}] &= 1 - \left(1 - \frac{1}{n}\right)^n \\ &\approx 1 - e^{-1} \\ &= 0.632 \end{aligned}$$

- Thus expected value of our e^1_{boot} is about $0.5 \times 0.368 = 0.184$.
- This is much less than the true value of 0.5.
- We can reduce this bias by doing what we did in cross-validation – for each model use only those data samples which are not used in learning it.

- So, we define the error estimate as

$$e_{boot} = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{|C^{-i}|} \sum_{b\in C^{-i}} L(\hat{f}^{b}(X_i), y_i)$$

  where $C^{-i} = \{b : X_i \notin b^{th}\ \text{bootstrap data set}\}$.
- We leave out any i for which C^{-i} is null.
- This is sometimes called the leave-one-out bootstrap estimate of error.

- In cross validation we divide data into at most n parts and hence can learn only n different models.
- In bootstrap we can choose B as high as we want and still have proper size of training sets.
- However, all training sets here are similar.
- Each bootstrap sample would have about $0.632\,n$ distinct samples.
- Hence, it roughly behaves like a 3-fold or 2-fold cross validation.
- Bootstrap is a general statistical technique and can be used in many estimation problems.

Bagging Estimates

- In bootstrap we generate many training sets by sampling from the original data.
- With these, we are learning B number of models, $\hat{f}^b$, $b = 1, \cdots, B$.
- These are used to get a good estimate of the expected error.
- Like in cross validation this allows us to assess a learning algorithm learning from a specific model class with n training samples.
- But in bootstrap we can learn any number of classifiers each on different subsets of the data.
- Can we use all these to improve prediction of final model?
- This is called Bagging.

- Bagging stands for Bootstrap Aggregation. It averages over the predictions of all models.
- Given all the bootstrap models learnt, at a point X, the *bagging* model prediction is

  $$\hat{f}_{bag}(X) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{b}(X)$$

  (Actually, this is the sample mean estimate (of size B) of the true bagging estimate).
- Often bagging can improve performance of the final model.

Combining Classifiers

- Bagging uses all models learnt on different (bootstrap) subsets of the data sets to predict on new data.
- Thus, bagging is an example of combining multiple classifiers to improve accuracy.
- This is the next topic we consider.

Combining Classifiers

- Classifier ensembles are often used to get better classifiers.
- Intuitively, using many classifiers should improve accuracy.
- 'Two heads are better than one'!
- It is like asking many experts and going by the 'majority' opinion.
- Bagging and Boosting are some general strategies for combining classifiers.

- If all the different classifiers are trained on the same data set we may not get much improvement.
- It is like having a committee of experts all with identical knowledge base.
- We want different classifiers in the ensemble to ‘complement’ each other.

- We have only one training set of data.
- Somehow we need to create multiple data sets (with appropriate variations) to train different classifiers.
- We can also think of introducing variability in the way a classifier is learnt.
- This is one of the main issues in designing classifier ensembles.
- That is, how to ensure variability in the different classifiers learnt.
- Another issue is how to combine the decisions of all the classifiers for prediction on new data.

- The individual classifiers are called base classifiers.
- Generally one uses the same type of base classifiers.
- Let us call a classifier ‘weak’ if it gives only a little more than 50% accuracy. (In a 2-class case).
- The idea of classifier ensembles is to combine many ‘weak’ classifiers so that what we get is a ‘strong’ one.

- In bagging, one relies on randomness to introduce variability in classifiers.
- A popular method is the so called *random forests*.
- Here the base classifiers are decision trees.

- In boosting, one relies on iterative reweighting of training samples to introduce variability in classifiers.
- A popular boosting algorithm is the so called Adaboost.

- We first consider a simple example to convey the idea of designing classifier ensembles.
- Suppose we want to create an ensemble with three component classifiers in a 2-class problem.
- Let $\mathcal{D}$ denote the given training data set with n points.
- We can begin by randomly selecting $n_1 < n$ points from $\mathcal{D}$, putting them in a set $\mathcal{D}_1$.
- We can learn our first component classifier h_1 using the training set $\mathcal{D}_1$.
- For learning the second component classifier h_2, what should be the data set?

- We may want a data set $\mathcal{D}_2$ that is most informative given h_1 in the sense that the performance of h_1 on $\mathcal{D}_2$ would be no better than pure guessing.
- So, we want roughly only half the points in $\mathcal{D}_2$ to be correctly classified by h_1
- So, we make $\mathcal{D}_2$ by choosing, with probability 0.5, those (remaining) points in $\mathcal{D}$ which are incorrectly classified by h_1 and with probability 0.5, we choose points correctly classified by h_1.
- Now we learn classifier h_2 using training set $\mathcal{D}_2$.

- We can think of the third component classifier to be a kind of tie-breaker of h_1 and h_2
- So, we can take all patterns in $\mathcal{D}$ where the classification by h_1 and h_2 differ and put it in a set $\mathcal{D}_3$.
- We now learn the third component classifier h_3 using training set $\mathcal{D}_3$.
- So, our third classifier specializes in resolving the cases where h_1 and h_2 differ.
- We use this classifier ensemble as follows.
- Given a new pattern X, if $h_1(X) = h_2(X)$ then that is what we output; otherwise we output $h_3(X)$.

- There are two issues in classifier ensembles
- We need to create some diversity in the training sets from which we learn component classifiers.
- In our example, we created this diversity through designing the training sets.
- The second issue is: how to combine the decisions of individual classifiers into the final decision.
- In the example, our combination of classifiers is a simple majority scheme; but still it is related to the way we learned the component classifiers.

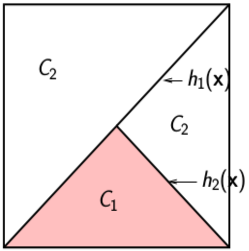
- As mentioned earlier, Bagging and Boosting are two generic approaches.
- We first briefly consider Bagging.
- In bagging we use multiple randomly generated bootstrap samples.
- Random Forests is an important method under bagging

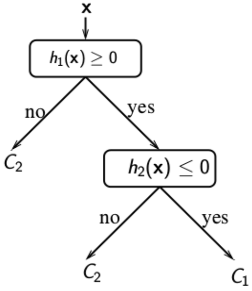
Random Forests

- Random forests is a method of learning a classifier ensemble using bagging.
- The base classifiers are decision trees.
- We learn many decision trees on randomly generated bootstrap data sets
- The decision tree learning is also randomized here.
- A majority rule is used for combining classifiers.
- Very simple method and is often very good in practice.

Decision Trees – Overview

- A decision tree is a very popular piece-wise linear classification method.
- Here is an example decision tree





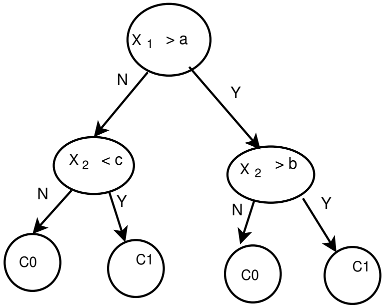
Decision Trees – Overview

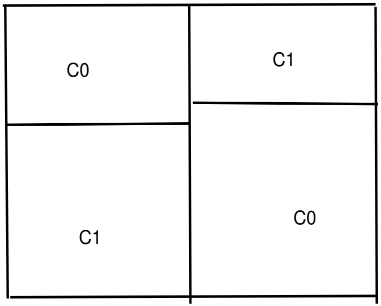
- Each (non-leaf) node is labelled with a so called ‘split rule’.
- All leaf nodes are labelled with a class label.
- Given a feature vector, at each node we go to one of the children based on the value of split rule on this feature vector.
- When we reach a leaf that gives the classification of the feature vector.

Decision Trees – Overview

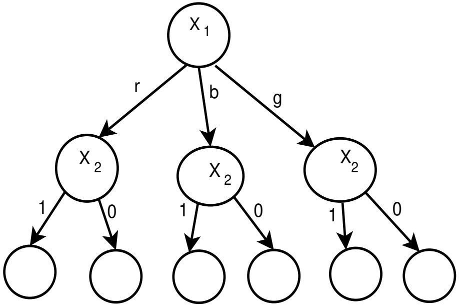
- If the split rule depends on only one of the features then the tree is termed axis-parallel; otherwise called oblique.
- If a feature takes only finitely many values, then the axis-parallel tree need not be binary.

Here is an example of axis parallel decision tree





An example of axis parallel decision tree with nominal features



- Learning a decision tree involves learning ‘best’ split rule at each node.
- Most decision tree algorithms learn the tree in a top-down fashion under a greedy heuristic.

- A top-down learning algorithm for decision trees learns the tree starting from the root.
- We start with the given training data. According to a chosen criterion we find the best split rule for this data.
- Then we make the root node with this split rule.
- We divide the data based on this split rule into the two sets that go into the left and right child nodes of the root.
- Now recursively we do the same thing at the two children nodes.
- We need a stopping criterion which decides when we decide to make a leaf node for the data.

We can write this as a recursive procedure (in pseudocode)

```
function GrowTree($S$)
Input: $S$, a set of samples, output: pointer to subtree
  1. If $S$ satisfies stopping criterion,
     then get a leaf node, label it with majority class of $S$ and return a pointer to it
  2. Else Find the best split rule for $S$. Get a node with this split rule and let $T$ point to it. Split $S$ into $S_l$ and $S_r$, the samples that go into the left and right child nodes based on the split rule found.
  3. left child of $T \leftarrow$ GrowTree($S_l$)
     right child of $T \leftarrow$ GrowTree($S_r$)
  4. return($T$)
```

Decision Trees – Overview

- A popular way of rating a split rule is by using the so called impurity measures.
- An impurity measure is intended to capture the 'skewness' of class distribution in a set of examples.
- We prefer split rules that make the class distribution in the left and right subtrees more skewed.

- Gini index is one such impurity measure.
- Let f be the mass function of a discrete random variable taking values $y_1, \cdots, y_k$. The so called gini index of this distribution is given by

$$\begin{aligned}
\text{gini}(f) &= \sum_{i,j=1}^{k} f(y_i) f(y_j) |y_i - y_j| \\
&= f(0)f(1) \; + \; f(1)f(0) \quad \text{for binary case} \\
&= (1 - (f(0))^2 - (f(1))^2) \quad \text{for binary case}
\end{aligned}$$

- Let S be a set of examples and let S_0 and S_1 be the subsets of examples of the two classes.
- Let $n_0 = |S_0|$, $n_1 = |S_1|$ and $n = |S|$. Then we take

$$\text{gini}(S) = \left(1 - \left(\frac{n_0}{n}\right)^2 - \left(\frac{n_1}{n}\right)^2\right)$$

- Let T be a split rule under which S gets split into S^l and S^r, the two sets that go to the left and right children.
- Let $|S^l| = n^l$ and $|S^r| = n^r$. Then

$$\text{gini gain}(T) = \text{gini}(S) - \left(\frac{n^l}{n}\,\text{gini}(S^l) + \frac{n^r}{n}\,\text{gini}(S^r)\right)$$

- We want T that gives maximum gini gain.

- Optimizing gini gain is not easy.
- If we have only nominal features then we can use an axis parallel non-binary tree
- We only need to decide which feature to use for the split rule and we can test all features to find one with best gini gain.
- If features are continuous, we can still use axis parallel trees and need to test only finitely many split rules.
- In general, one uses random search methods.
- There are many different criteria other than gini gain for rating split rules.

- The Random Forests algorithm employs axis parallel trees.
- It uses multiple training sets obtained as bootstrap samples
- It uses randomness in tree-learning also: e.g., choosing randomly among the top few split rules
- Randomness is used in stopping criterion also.
- Final prediction is based on a majority decision

Boosting

- Boosting is another method to combine many (weak) classifiers to produce a much better one.
- We learn multiple classifiers from the given training data.
- In boosting we introduce variation in the classifier ensemble by assigning different weights to samples in the training data while learning different classifiers.

- AdaBoost is a very popular boosting algorithm.
- We will consider the AdaBoost for 2-class classification

- In AdaBoost we assign (non-negative) weights to points in the data set.
- The algorithm iteratively keeps learning new classifiers.
- In each iteration, we learn a classifier to minimize **weighted error** on training data.
- We assume we can find a classifier with weighted error less than 50%.
- After learning the current classifier, we increase the (relative) weights of data points that are misclassified by the current classifier.
- We learn another classifier with the modified weights.
- The variability in the classifier ensemble comes from the modification of weights.
- The final classifier is a weighted majority voting by all the classifiers.

Notation

- Let $\{(X_1, y_1), \cdots, (X_n, y_n)\}$ be the data. We take $y_i \in \{-1,\ +1\}$.
- Let $w_i(k)$ denote the weight for the i^{th} data point at k^{th} iteration.
- Let h_k denote the classifier learned at k^{th} iteration; we take $h_k(X) \in \{-1,\ +1\}$.
- We assume that the error rate of each classifier on its training data is less than 0.5.
- AdaBoost is meant to combine an arbitrary number of such weak classifiers to construct a strong classifier.

AdaBoost Algorithm

1. Initialize: $w_i(1) = \frac{1}{n}, \ \forall i$.
2. For m=1 to M do
   a. Learn classifier h_m to minimize $\sum_{i=1}^{n} w_i(m) I_{[y_i \neq h_m(X_i)]}$ where I_A is an indicator of A.
   b. Let
   $$\xi_m = \sum_{i=1}^{n} w_i(m) I_{[y_i \neq h_m(X_i)]}$$
   (We assume $\xi_m < 0.5$).
   c. Set $\alpha_m = \frac{1}{2} \ln \left(\frac{1-\xi_m}{\xi_m} \right)$. (We have $\alpha_m > 0$).

Algorithm contd.

d. Update the weights by

$$\begin{aligned} w'_i(m+1) &= w_i(m)\ \exp\left(-\alpha_m\ y_i\ h_m(X_i)\right) \\ w_i(m+1) &= \frac{w'_i(m+1)}{\sum_i w'_i(m+1)} \end{aligned}$$

3. Output the final classifier:

$$h(X) = \mathsf{sgn}\left(\sum_{m=1}^{M} \alpha_m\ h_m(X)\right)$$

- Each h_m is called a base classifier or component classifier.
- The algorithm to learn them is the base learning algorithm.
- We can use any method as base learning algorithm.
- In this sense, Adaboost is a meta-learning algorithm.
- M is a parameter. We can continue the procedure for arbitrary number of iterations.

- In each iteration we update weights as

$$\begin{aligned} w_i'(m+1) &= w_i(m)\, \exp\left(-\alpha_m\, y_i\, h_m(X_i)\right) \\ w_i(m+1) &= \frac{w_i'(m+1)}{\sum_i w_i'(m+1)} \end{aligned}$$

- If $h_m(X_i) \neq y_i$, then $w_i(m+1) > w_i(m)$.
- If the current classifier misclassifies a pattern, its weight for the next iteration is increased.

- But if we keep arbitrarily increasing weights of misclassified patterns, then subsequent training sets will be predominantly only those patterns.
- We want only about half the weightage for samples misclassified by current classifier.
- The weight update scheme of AdaBoost ensures

$$\sum_{i\,:\,h_m(X_i)\neq y_i} w_i(m+1) = \sum_{i\,:\,h_m(X_i)=y_i} w_i(m+1) = \frac{1}{2}$$

- We can show this as follows.

Recall the weight update formulas:

$$\begin{aligned}
\xi_m &= \sum_{i=1}^{n} w_i(m) I_{[y_i \neq h_m(X_i)]} \\
\alpha_m &= \frac{1}{2} \ln \left(\frac{1-\xi_m}{\xi_m} \right) \\
w_i'(m+1) &= w_i(m)\ \exp\left(-\alpha_m\ y_i\ h_m(X_i)\right) \\
w_i(m+1) &= \frac{w_i'(m+1)}{\sum_i w_i'(m+1)}
\end{aligned}$$

- By definition, we have

$$\xi_m = \sum_{i\,:\,h_m(X_i)\neq y_i} w_i(m)$$

- Since the weights are always normalized, we have

$$\sum_{i\,:\,h_m(X_i)=y_i} w_i(m) = 1 - \xi_m$$

- Also, by definition, $\exp(\alpha_m) = \sqrt{\frac{1-\xi_m}{\xi_m}}$

Now we have

$$\begin{aligned}
\sum_i w_i'(m+1) &= \sum_i w_i(m) \exp\left(-\alpha_m \; y_i \; h_m(X_i)\right) \\
&= \sum_{h_m(X_i) \neq y_i} w_i(m) e^{\alpha_m} + \sum_{h_m(X_i) = y_i} w_i(m) e^{-\alpha_m} \\
&= e^{\alpha_m} \sum_{h_m(X_i) \neq y_i} w_i(m) + e^{-\alpha_m} \sum_{h_m(X_i) = y_i} w_i(m) \\
&= \xi_m \sqrt{\frac{1-\xi_m}{\xi_m}} + (1-\xi_m)\sqrt{\frac{\xi_m}{1-\xi_m}} \\
&= 2\sqrt{(1-\xi_m)\xi_m}
\end{aligned}$$

We also have

$$\begin{aligned}
\sum_{h_m(X_i)\neq y_i} w_i'(m+1) &= \sum_{h_m(X_i)\neq y_i} w_i(m)\exp\left(-\alpha_m\, y_i\, h_m(X_i)\right) \\
&= \exp(\alpha_m) \sum_{h_m(X_i)\neq y_i} w_i(m) \\
&= \exp(\alpha_m)\, \xi_m \\
&= \sqrt{(1-\xi_m)\xi_m}
\end{aligned}$$

Using this, we get

$$\begin{aligned}
\sum_{h_m(X_i)\neq y_i} w_i(m+1) &= \sum_{h_m(X_i)\neq y_i} \frac{w_i'(m+1)}{\sum_i w_i'(m+1)} \\
&= \frac{\sum_{h_m(X_i)\neq y_i} w_i'(m+1)}{\sum_i w_i'(m+1)} \\
&= \frac{\sqrt{(1-\xi_m)\xi_m}}{2\sqrt{(1-\xi_m)\xi_m}} \\
&= \frac{1}{2}
\end{aligned}$$

- Thus we have shown that

$$\sum_{i\,:\,h_m(X_i)\neq y_i} w_i(m+1) = \frac{1}{2}$$

- The weight update is such that at the next iteration, half the total weight is for patterns misclassified by the current classifier and the remaining half is for patterns correctly classified by the current classifier.
- This gives correct level of variability in successive classifiers that are learnt.

- The final classification decision on a new X by this classifier ensemble is

$$h(X) = \text{sgn}\left(\sum_{m=1}^{M} \alpha_m \, h_m(X)\right)$$

- If α_m are same for all m, this is a simple majority decision. (Recall $h_m(X) \in \{-1, \, +1\}$).
- Here each component classifier has a different weight for its vote.
- The weight is based on the accuracy of that classifier.

- This algorithm can generate a classifier ensemble with any number of component classifiers.
- The component classifiers can be of any type.
- For each classifier we only require that $\xi_m < 0.5$.
- The strength of this boosting techniques is that we can combine many such weak classifiers to finally come up with a classifier with high accuracy.

Base Learning Algorithm

- The base learning algorithm can be anything.
- However, base classifier should minimize weighted error.
- Most learning algorithms minimize errors in training set.
- Here training samples have weights.
- How does one design classifiers to minimize weighted errors?

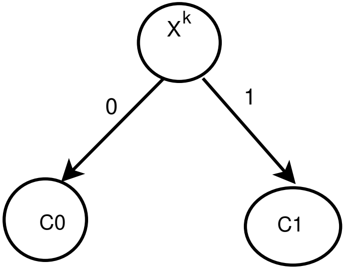
- One way is to generate different random training sets for each iteration.
- At m^{th} iteration, we generate a random set of examples by sampling with replacement from the original training set using the distribution $\{w_i(m)\}$.
- Now an algorithm minimizing errors on this random training set (approximately) minimizes the weighted error as needed.
- This is one of the standard ways of using AdaBoost.
- Called boosting by resampling.

- We can also design an algorithm to minimize weighted errors on training set.
- Note that the individual classifiers need not achieve high accuracy.

A Simple Base learning Algorithm

- A simple method to minimize weighted training error is to learn the best decision stump.
- A decision stump is a two-level decision tree.
- At the root, we test a chosen feature and based on its value take one of the branches.
- All second level nodes are leaves with class labels.
- We take class labels to be $+1$ and -1.

- Let us consider learning the best decision stump with respect to k^{th} feature, X^k, which is binary:



- Every decision stump is the classifier:

$$\begin{aligned} h(X) &= c_0 \text{ if } X^k = 0 \\ &= c_1 \text{ if } X^k = 1 \end{aligned}$$

- Our task is to find optimal values for c_0 and c_1

- Suppose we calculate for $j \in \{0, 1\}$ and $b \in \{+1, -1\}$,

$$W_b^j = \sum_{i:X_i^k = j \& y_i = b} w_i$$

- W_{+1}^0 would be the sum of weights of all examples in class $+1$ whose k^{th} component is 0.
- We can compute W_b^j in linear time over number of examples.

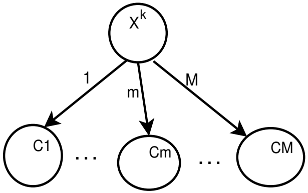
- For our h we need to find optimal values of c_0 and c_1 to minimize weighted error.
- It is easy to see that the best values are

$$\begin{aligned} c_0 &= +1 \text{ if } W^0_{+1} \geq W^0_{-1} \\ &= -1 \text{ if } W^0_{-1} > W^0_{+1} \end{aligned}$$

- That is, if X^k takes value 0, then we are better off putting X in class $+1$ if this feature is 0 "more often" in class $+1$.
- Similarly we can decide best value for c_1.
- This gives us the best decision stump we can get by splitting on k^{th} feature.
- Note that $W^0_{-c_0} + W^1_{-c_1}$ is the weighted error of our h.

- Now suppose X^k takes values in $\{1, \cdots, M\}$.
- The structure of decision stump classifier is

$$h(X) = c_m \ \text{ if } \ X^k = m, \ m = 1, \cdots, M.$$



- We now compute W_b^j for $j \in \{1, \cdots, M\}$ and $b \in \{+1, -1\}$.
- Specifying h is fixing c_m (as either $+1$ or -1) for each m.
- The best h is given by the same equation as earlier:

$$\begin{aligned} c_m &= +1 \ \text{ if } \ W_{+1}^m \geq W_{-1}^m \\ &= -1 \ \text{ if } \ W_{-1}^m > W_{+1}^m \end{aligned}$$

- Now suppose X^k is real valued.
- Then the structure of decision stump classifier is

$$\begin{aligned} h(X) &= c_0 \text{ if } X^k \geq \tau \\ &= c_1 \text{ if } X^k < \tau \end{aligned}$$

- For a given τ this is like a binary feature.
- Given n examples, there are only $n+1$ distinguishable values for τ.
- So, we can find the best decision stump for k^{th} feature when it is real-valued also.

- Thus, we can find best decision stump using any one of the features.
- We know the weighted error rate of each of them.
- Hence we can find the best decision stump classifier to minimize weighted error considering all the features.