Overfitting
Cross Validation
Mixture of Gaussians
ROC Curves
Apriori Algorithm
AdaBoost

Lecture # 16

Introduction

Sometimes when minimizing the sum of the squares to find the best fit, we may run into overfitting problem. This is when a set of model parameters give an excellent fit to the data with small error but when the same model is applied on new data, the prediction is not that accurate. In other words, overfitting is defined as follows:

Given a hypothesis space H, a hypothesis $h \in H$ is said to overfit the training data if there exist some alternative hypothesis $h' \in H$ such that h has smaller error than h' over the training sample but h' has a smaller error than h over the entire distribution of data.

Generalization in Machine Learning

The learning of the target function from training data in Machine Learning is called inductive learning. This refers to learning general concepts from specific examples which is exactly the problem that supervised ML problems aim to solve.

Generalization refers to how well the concepts learned by a ML model apply to specific examples not seen by the model when it was learning.

A ML model generalizes well from the training data to any data from the problem domain, allowing us to make predictions in the future on data the model has never seen.

Overfitting (or underfitting) is how well a ML model learns and generalizes to new data.

Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize.

Overfitting is more likely with nonparametric and nonlinear models that have more flexibility when learning a target function. As such, many nonparametric machine learning algorithms also include parameters or techniques to limit and constrain how much detail the model learns.

Causes of Overfitting and Solutions

Overfitting could be caused by a number of things:

1. if the problem is insufficiently constrained: for example, if we have ten measurements and ten model parameters, then we can often obtain a perfect fit.

2. Fitting noise: overfitting can occur when the model is so powerful that it can fit the data and the random noise in the data.

There are two important solutions to the overfitting problem: adding prior knowledge and handling uncertainty. In many cases there is some prior knowledge we can leverage. A very common assumption is that the underlying function is likely to be smooth-i.e. having small derivatives. Also, assuming smoothness reduces model complexity as it is easier to estimate smooth models from small datasets.

To solve overfitting problem we add regularization. This is an extra term to the learning objective function that prefers smooth models. For regressions and for many multi-dimensional functions, this can be done with a function of the form:

$$E(w) = \|y - Bw\|^2 + \lambda \|w\|^2$$

Where the first is the data term that measures the model fit to the training data. The second term is the *smoothness* term that penalizes non-smoothness (rapid change in $f(x)$). This smoothness is called weight decay because it tends to make the weight smaller. Rapid changes in the slope of function f (i.e. high curvature) can only be created in regressions by adding and subtracting basis functions with large weights. In fact, we may directly penalize smoothness by using a term that directly penalizes the integral of the squared curvature of f(x).

We can write this as follows:

$E(w) = (y - B\,w)^T (y - B\,w) + \lambda\, w^T w = w^T B^T B\, w - 2\, w^T B^T y + \lambda\, w^T w + y^T y$

$= w^T (B^T B + \lambda I)\, w - 2\, w^T B^T y + y^T y$

To minimize $E(w)$, we solve the normal equation

$$\nabla E(w) = 0 \text{ or } \frac{\partial E}{\partial w_i} = 0$$

For all *i values. This gives the regularized solution for w*

$w^* = (B^T B + \lambda I)^{-1} B^T y$

How to Reduce uncertainties due to overfitting?

Overfitting is a problem because the evaluation of machine learning algorithms on training data is different from the evaluation we actually care the most about, namely how well the algorithm performs on unseen data. There are two techniques that are used to limit overfitting:

1. Use a resampling technique to estimate model accuracy.

2. Hold back a validation dataset.

The most popular resampling technique is k-fold cross validation. It allows y training and testing a model k-times on different subsets of training data to build up an estimate of the performance of a machine learning model on unseen data.

A validation dataset is simply a subset of the training data that you hold back from your machine learning algorithms until the very end of your project. After you have selected and tuned your machine learning algorithms on your training dataset you can evaluate the learned models on the validation dataset to get a final objective idea of how the models might perform on unseen data.

Sources used for this lecture

Jason Brownlee in understanding machine learning algorithm

https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/

Cross Validation

Definition

How do we choose between two possible ways to fit data?

Simply measuring how well a model fits data would mean that we always try to fit the data as closely as possible. However, fitting the data is no guarantee that we will be able to generalize to new measurements. For example, consider the use of polynomial regression to model a function given a set of data points. Higher-order polynomials will always fit the data as well or better than a low-order polynomial; indeed, an N − 1 degree polynomial will fit N data points exactly (to within numerical error). So just fitting the data as well as we can, does not necessarily guarantee that it could be generalized to all cases of interest. The general solution is to evaluate models by testing them on a new data set (the “test set”), distinct from the training set. This measures how predictive the model is: i.e. Is it useful in new situations? More generally, we often wish to obtain empirical estimates of performance. This can be useful for finding errors in implementation, comparing competing models and learning algorithms, and detecting over or under fitting in a learned model.

A simple way of doing this is to partition our data into two sets- training set and validation set. Let K be the unknown model parameter. We pick a range of values for K. Then, for each possible value of K, we learn a model on that K on the training set and compute that model's error on the validation set. The error on the validation set could just be

$$\sum_i \|y_i - f(x_i)\|^2$$

We then pick the K which has the smallest validation set error. The problem with this method is that we only get reliable result if our initial training set is large.

The solution to this is **N-fold cross validation**. In this approach we randomly partition the training data into N sets of equal size and run the learning algorithm N times. Each time, a different one of the N sets will be the test set and the model is trained on the remaining N-1 sets. The value of K is the average of the errors across the N test errors. We then pick the value of K that has the lowest score and then learn model parameters for this K.

Sources used for this lecture

This section is taken from Aaron Hertzmann and David Fleet lecture notes in Machine Learning- University of Toronto

K-Means Clustering: Mixture of Gaussians

MoG Definition

The Mixture of Gaussians (MoG) model is a generalization of K-means clustering. While K-means works for clusters that are more or less spherical, the MoG model can handle other cluster shapes or even overlapping clusters. This is a probabilistic measure of the clustering.

The MoG model consists of K Gaussian distributions, each with their own means and covariances (μ_j, K_j). Each Gaussian also has an associated probability, a_j, which represents the fraction of the data that are assigned to different Gaussian components. The idea is that each Gaussian component in the mixture should correspond to a single cluster (Figure 3). The model parameters can be written as $\theta = \{a_{\{1,K\}}, \mu_{\{1,K\}}, K_{\{1,K\}}\}$.

MoG Formulation

The probabilistic model comprise the probabilities of each Gaussian component and Gaussian likelihood over the data space for each component. We can write

$$P(L = j \mid \theta) = a_j$$
$$p(y \mid \theta, L = j) = G\,(y;\ \mu_j, K_j)$$

To sample a single data point from this model, we first randomly select a Gaussian component according to their probabilities *aj* and then we randomly sample from the corresponding Gaussian component. We can write

$$p(y|\theta) = \sum_{j=1}^{K} p(y, L = j \mid \theta)$$
$$= \sum_{j=1}^{K} p\,(y \mid L = j, \theta)\, P(L = j \mid \theta)$$
$$= \sum_{j=1}^{K} a_j \frac{1}{\sqrt{(2\pi)^D K_j}} \exp[-\frac{1}{2}(y - \mu_j)^T K_j^{-1}\,(y - \mu_j)]$$

Where D is the dimension of the data. This model can be interpreted as linear combination of Gaussians. We get a multi modal Gaussians by adding together unimodal Gaussians.

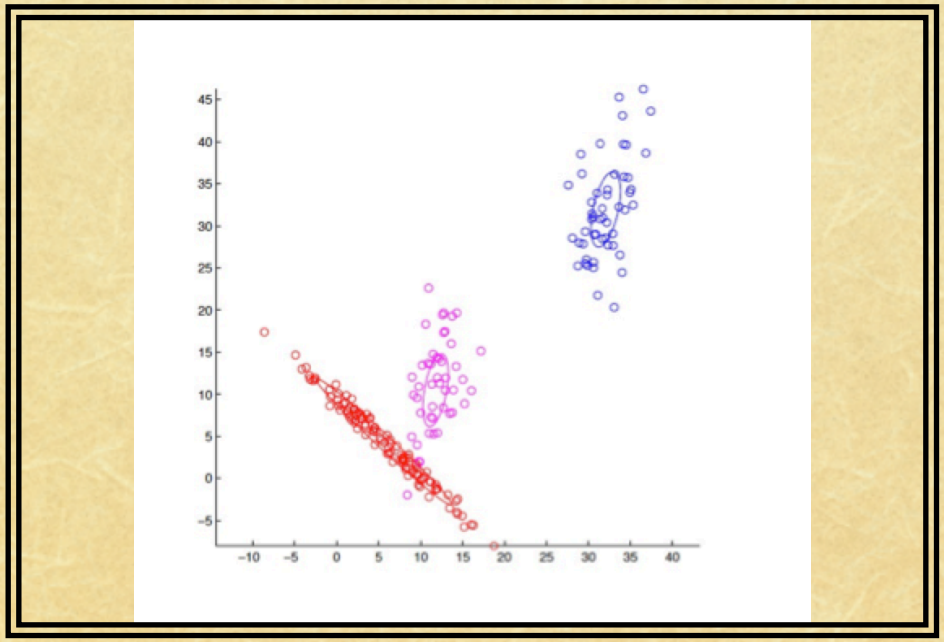


Figure 3: Shows examples of clustering using MoG method. This classifies data points in both spherical clusters and those with flat distribution.

Sources used for this lecture

This section is taken from Aaron Hertzmann and David Fleet lecture notes in Machine Learning- University of Toronto

ROC Curves

Receiver Operating Characteristic (ROC) Curve

A **Receiver Operating Characteristic** curve (ROC) is a plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the True Positive rate (TPR) against the False Positive Rate (FPR) at various threshold settings (Figure 4).

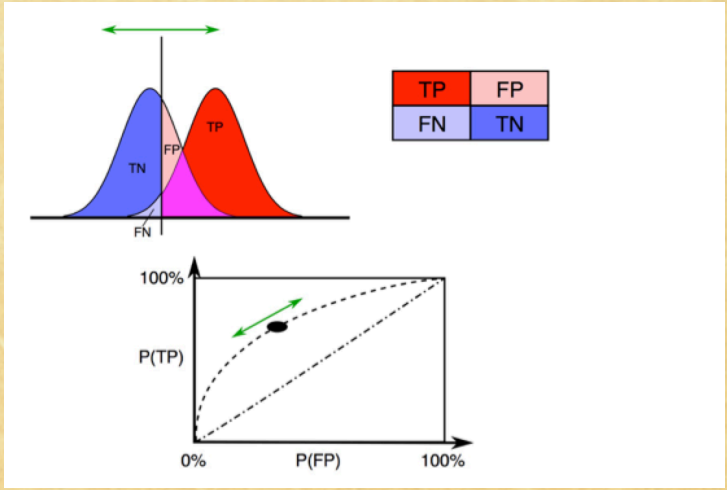
Let us consider a two-class prediction problem (binary classification), in which the outcomes are labeled either as positive (p) or negative (n). There are four possible outcomes from a binary classifier. If the outcome from a prediction is p and the actual value is also p, then it is called a true positive (TP); however if the actual value is n then it is said to be a false positive (FP). Conversely, a true negative (TN) has occurred when both the prediction outcome and the actual value are n, and false negative (FN) is when the prediction outcome is n while the actual value is p.

To draw a ROC curve, only the true positive rate (TPR) and false positive rate (FPR) are needed (as functions of some classifier parameter). The TPR defines how many correct positive results occur among all positive samples available during the test. FPR, on the other hand, defines how many incorrect positive results occur among all negative samples available during the test.

A ROC space is defined by FPR and TPR as x and y axes, respectively, which depicts relative trade-offs between true positive (benefits) and false positive (costs)- (Figure 5). Since TPR is equivalent to sensitivity and FPR is equal to 1 – specificity, the ROC graph is sometimes called the sensitivity vs (1 – specificity) plot.

Two independent probability distributions are shown in red and blue. They represent “True Positive (TP)” (red), “True Negative (TN)” (blue), “False Positive (FP) (pink)” and “False Negative (FN) (light blue)”. This provides the likelihood that a classification is correct.

Lower panel: Probability of TP vs. FP (ROC Diagram) showing the fraction of TP and FP classifications. The best classification is when P(TP) = 100%.



Quantitative Measure of Classification Performance

The four cases (TP, TN FP and FN) can be used to define several terms that are useful in measuring classification performance, as described below:

Sensitivity: is the ability of a classifier to select all the cases that need to be selected. A perfect classifier will select all the actual Y's. It will have no FNs. Sensitivity is defined as the ratio (or percentage) TP/(TP+FN). However, sensitivity alone is not sufficient to evaluate a classifier. The TNs are also needed.

Specificity: is the ability of a classifier to reject all the cases that need to be rejected. A perfect classifier will reject all the actual N's and will not deliver any unexpected results. It will have no FPs. Specificity is expressed as the ratio TN/(TN+FP).

Relevance: suppose a search is done to identify specific terms and that returned 100 documents of which 70 were relevant. Furthermore, the search missed out on a additional 40 documents that could have been useful.

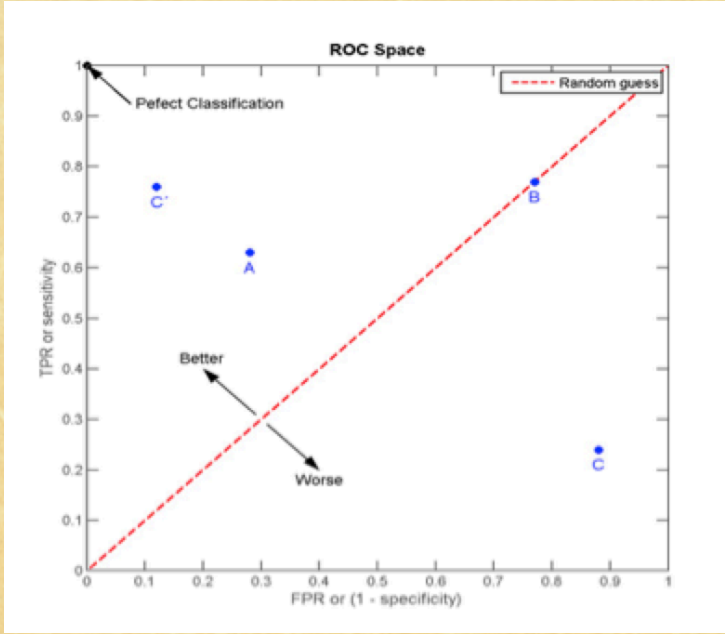
Precision is defined as the proprtion of cases found that were relevant. From the above case, the number was 70/100 = 70%. The 70 documents were TP while 30 were FP. Therefore precision is defined as TP/(TP+FP).

Recall: is defined as the proportion of the relevant cases that were found among all the relevant cases. In the above example, only 70 of the total of 110 (70 found+40 missed) relevant cases were actually found, giving a recall 70/110 = 63.63%. This is defined as TP/(TP+FN)

Accuracy: is defined as the ability of the classifier to select all cases that need to be selected and reject all cases that need to be rejected. For a classifier with 100% accuracy, this would imply that FN=FP=0. Accuracy is given by (TP+TN)/(TP+FP+TN+FN).

Error: is the complement of accuracy defined as (1 – Accuracy)

ROC diagram shows classifications of different data points. perfect classification.



Sources used for this lecture

Machine Learning in Action

Peter Harrington

Data Science – Concepts and Practice

Vijay Kotu and Bala Deshpande

Association Analysis with the Apriori Algorithm

Definitions

Looking at hidden relationships in large datasets is known as Association Analysis. The problem is that finding different combinations of items can be time consuming and expensive in terms of computing time. Apriori Algorithm will solve this problem.

The interesting relationships can take two forms: frequent item sets or association rules. Frequent item sets are a collection of items that frequently occur together. Association rule suggests that a strong relation exists between two items.

Example

The following table lists a number of transactions:

Transaction #	Items
0	soy milk, lettuce
1	Lettuce, diapers, wine, chard
2	Soy milk, diapers, wine, orange juice
3	Lettuce, soy milk, diapers, wine
4	Lettuce, soy milk, diapers, orange juice

From this table we can find an association rule such that diaper -> wine which means that if people buy diaper, it is a good chance that they will buy wine as well. With this information retailers develop a good idea of their customers. They will group special items together. This will be applicable to other industries such as web-site traffic analysis and medicine.

Support and Confidence

How do we define the "interesting relationships"? What is the definition of "frequent"? Who defines what is "interesting"?

There are two concepts that we could use to select these: support and confidence.

Support: the support of an item is defined as the percentage of the data set that contains these items. From the last example, the support of soy milk is 4/5 while the support of [soy milk, diaper] is 3/5 because of out of 5 transactions, 3 contain both soy milk and diapers. In this case we can define a minimum support and get only the items that meet that minimum support.

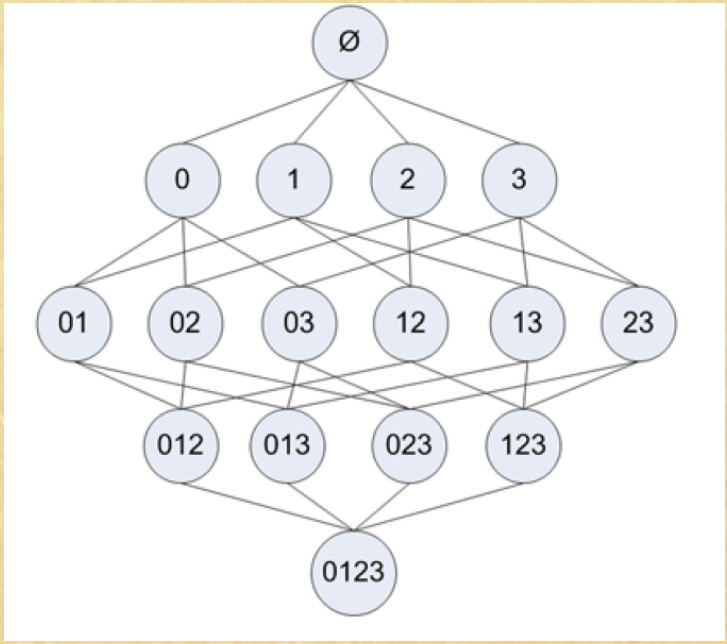
Confidence: this is defined for an association rule like [diapers] -> [wine]. From the above example, the confidence for this rule is defined as support(diapers, wine/support(diapers). The support of [diapers, wine] is 3/5. The support of diapers is 4/5 so the confidence for diapers → wine is ¾ = 0.75. This means that in 75% of the items in our datasetcontaining diapers, our rule is correct (meaning diapers are associated with wine).

With support and confidence, we can quantify the success of our association analysis. Now, imagine that we want to find all sets of items with a support greater than 0.8. We could generate a list of every combination of items and then count how frequently that occurs. However, this process is very slow and expensive in terms of computing time when applied on large datasets. Here we use the Apriori Principle.

Apriori Principle

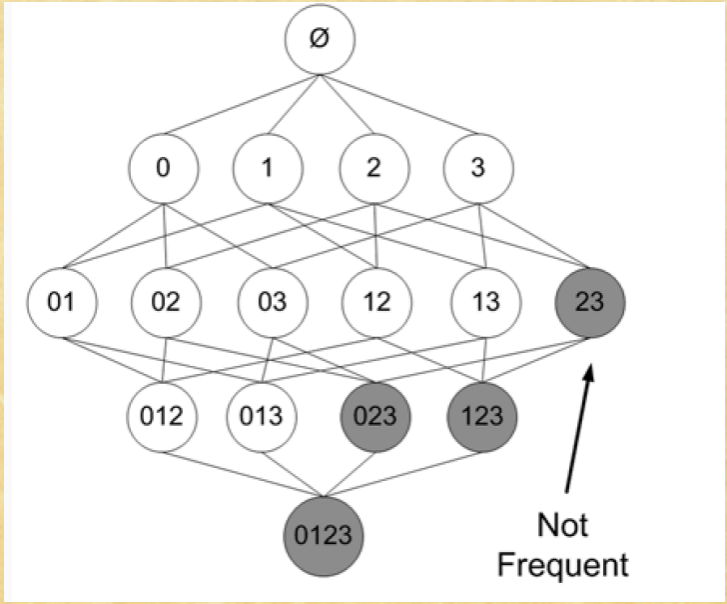
Lets assume there are four items in a store: 0, 1, 2, 3. What are all the possible combinations of these items that could be purchased?

The diagram shows all possible combinations of the items. Our goal is to find sets of items that are purchased together frequently. We measure frequency by the support of a set. The support of a set counted the percentage of transactions that contained that set. For example, to find support of a given set {0,3}, we go through all the combinations and identify those containing both 0 and 3. we then divide this by the total number of transactions. This gives our support. We repeat this for all the sets. In this case we need to go through this 15 times to count all the combinations.



This number gets large very quickly. A data set that contains N possible items can generate $2^N - 1$ possible item sets. A store selling 100 items can have $1.26*10^{30}$ combinations. This will soon become very complicated.

To reduce the time needed to compute this, Apriori Principle is developed. This helps us to reduce the number of interesting items. The Apriori Principle states that if an item set is infrequent, then any item set containing that will also be infrequent. For example, if item set {2,3} is infrequent, from this knowledge item sets {0,2,3}, {1,2,3} and {0,1,2,3} will all be infrequent. Using this, we can slow down the growth of our exponential increase of item sets and calculate the list of frequent item sets.



AdaBoost Meta-algorithm for Improving Classifications

Improving Classification with AdaBoost Meta-algorithm

Meta-algorithms are a way of combining other algorithms. We'll focus on one of the most popular meta-algorithms called AdaBoost. AdaBoost is considered to be the best-supervised learning algorithm.

You've seen different algorithms for classification. These algorithms have individual strengths and weaknesses. One idea that naturally arises is combining multiple classifiers.

Methods that do this are known as ensemble methods or meta-algorithms. Ensemble methods can take the form of using different algorithms, using the same algorithm with different settings, or assigning different parts of the dataset to different classifiers.

Building classifiers from randomly resampling data

Bootstrap aggregating, which is known as bagging, is a technique where the data is taken from the original dataset S times to make S new datasets. The datasets are the same size as the original. Each dataset is built by randomly selecting an example from the original with replacement. By "with replacement" we mean that you can select the same example more than once. This property allows you to have values in the new dataset that are repeated, and some values from the original won't be present in the new set.

After the S datasets are built, a learning algorithm is applied to each one individually.

When we like to classify a new piece of data, we apply our S classifiers to the new piece of data and take a majority vote.

Boosting

Boosting is a technique similar to bagging. In boosting and bagging, you always use the same type of classifier. But in boosting, the different classifiers are trained sequentially. Each new classifier is trained based on the performance of those already trained. Boosting makes new classifiers focus on data that was previously misclassified by previous classifiers.

Boosting is different from bagging because the output is calculated from a weighted sum of all classifiers. The weights aren't equal as in bagging but are based on how successful the classifier was in the previous iteration. There are many versions of boosting, the most popular being Adaptive Boosting- AdaBoost.

Training and Improving the Classifier

An interesting theoretical question is: can we take a weak classifier and use multiple instances of it to create a strong classifier? By "weak" we mean the classifier does a better job than randomly guessing but not by much. That is to say, its error rate is greater than 50% in the two-class case. The "strong" classifier will have a much lower error rate. The AdaBoost algorithm was born out of this question (Figure 1).

AdaBoost is short for adaptive boosting. AdaBoost works this way: A weight is applied to every example in the training data. We'll call the weight vector D. Initially, these weights are all equal. A weak classifier is first trained on the training data. The errors from the weak classifier are calculated, and the weak classifier is trained a second time with the same dataset. This second time the weak classifier is trained, the weights of the training set are adjusted so the examples properly classified the first time are weighted less and the examples incorrectly classified in the first iteration are weighted more. To get one answer from all of these weak classifiers, AdaBoost assigns α value to each of the classifiers. The values are based on the error of each weak classifier.

The error ε is given by

$$\varepsilon = \frac{number\ of\ incorrectly\ classified\ examples}{total\ number\ of\ examples}$$

$$\alpha = \frac{1}{2} ln(\frac{1-\varepsilon}{\varepsilon})$$

After we calculate α, we can update the weight vector D so that the examples that are correctly classified will decrease in weight and the misclassified examples will increase in weight. D is given by

$$D_i^{t+1} = D_i^t \frac{e^{-\alpha}}{Sum(D)} \quad if\ correctly\ predicted$$

$$D_i^{t+1} = \frac{D_i^t e^{\alpha}}{Sum(D)} \quad if\ incorrectly\ classified$$

After D is calculated, AdaBoost starts on the next iteration. The AdaBoost algorithm repeats the training and weight-adjusting iterations until the training error is 0 or until the number of weak classifiers reaches a user-defined value.

Implementation of the full Adaboost algorithm

We built a classifier that could make decisions based on weighted input values. We now have all we need to implement the full AdaBoost algorithm.

Pseudo-code for this will look like this:

For each iteration:

Find the best stump using buildStump()

Add the best stump to the stump array

Calculate alpha

Calculate the new weight vector – D

Update the aggregate class estimate

If the error rate ==0.0 : break out of the for loop

A schematic diagram showing combination of results from different classifiers with different weights (α values) shown in triangles. The combined result is shown as Σ.

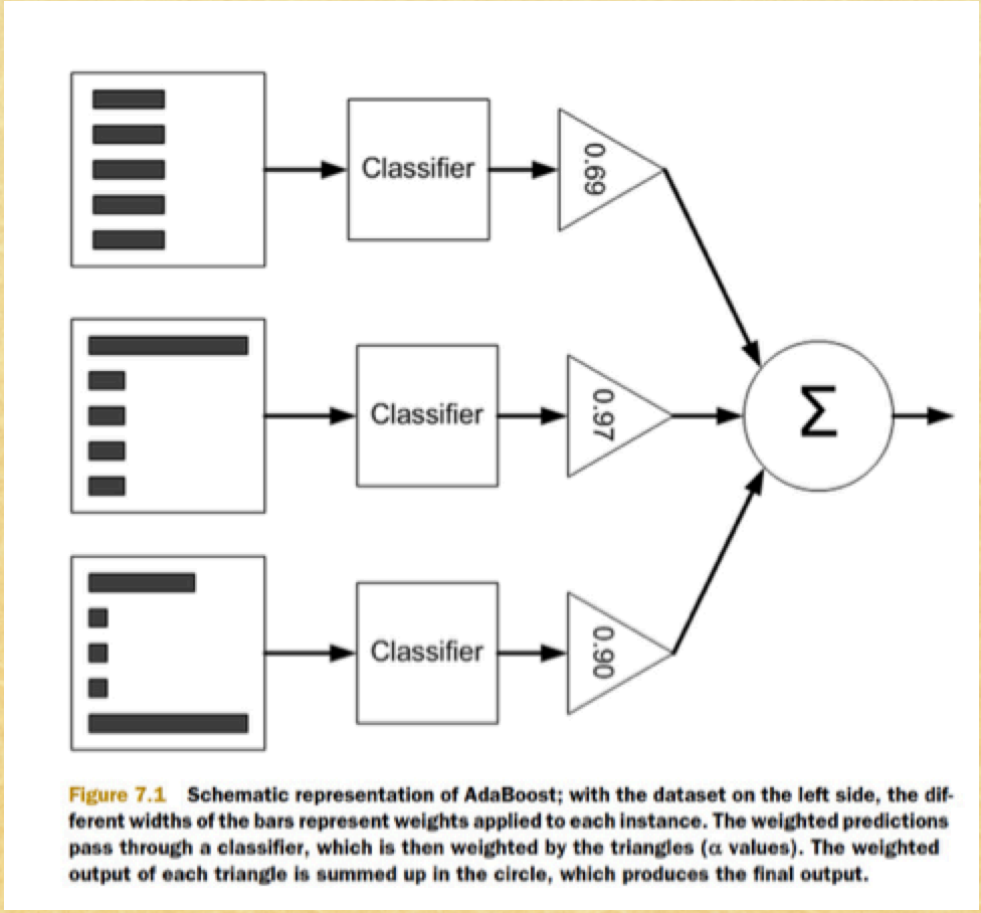


Figure 7.1 Schematic representation of AdaBoost; with the dataset on the left side, the different widths of the bars represent weights applied to each instance. The weighted predictions pass through a classifier, which is then weighted by the triangles (α values). The weighted output of each triangle is summed up in the circle, which produces the final output.