Ensemble Methods

Andrés Ponce

May 10, 2020

Ensemble Methods refer to when we use an aggregate of L different models to find more accurate predictions for our data. There are several ways of implementing a selection from several methods: adaboost, decision trees, bagging, random forests, etc...

Boosting

The straightforward approach to using multiple models is to take the average value of each of the models that we use. Supposing we want to use M models, and we want to predict the value given by a certain function, we could say

$$y_{COM} = \frac{1}{M} \sum_{m=1}^{M} y_m(x)$$

where COM refers to the *decision by committee* given by the equations. We take the average value of the M models to form our final result. When calculating the error, we also take the aggregate error of all the models.

Boosting then refers to the building of a strong classifier for a problem by using several weaker classifiers. When boosting, we again can have M models whose scores are summed up.

The type of boosting called **Adaptive Boosting** will pass the input through the models sequentially, and *train* the models sequentially. Then, when training the next model, items that were misclassified will be given greater weight. This may result in an overall model which performs very well even if the models themselves perform as well as random ones.

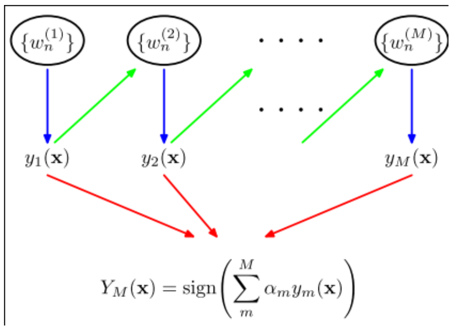

Figure 1: Each of the models contributes to how we pass the input set to the next model to be trained. At the end, we also use each model's results to calculate the final prediction.

The process for AdaBoost looks something like:

1. Given inputs $(x_1, y1), ..., (x_m, y_m)$ where $x_i \in X, y_i \in \{-1, +1\}$, initialize their distributions $D_t(i) = \frac{1}{m}, i = 1, ..., m$ [1]

2. Find the classifier h_t which minimizes the error w.r.t. D_t.[2]

3. Calculate the weight classifier $\alpha_t = \frac{1}{2} ln \frac{1-\epsilon_t}{\epsilon t}$

4. Update our distribution [3]

$$D_{t+1}(i) = \frac{D_t \exp[-\alpha_t y_i h_t(x_i)]}{Z_t}$$

Once we have the final classifier $H(x)$, comprised of the sum of the modesl with the adjusted weights and distributions, we can calculate

[1] The distributions $D_t(i)$ mean the probability distribution of x_i's at time t

[2] We can basically take the min of

$$\epsilon_j = \sum_{i=1}^{m} D_t(i)[y_i \neq h_j(x_i)]$$

[3] Z_t is basically a value to normalize the distribution.

the **margin** of the classifier on a sample (x, y). The margin is given by [4]

$$yH(x)$$

[4] The margin tells us when we classified the input correctly, and our confidence with our classification.

The AdaBoost algorithm will try to maximize the margins of our variables by minimizing the **exponential loss** of the predictions.

$$loss_{exp}[H(x)] = E_{x,y}[e^{-yH(x)}]$$

Face Detection using AdaBoost

To perform facial detection on an image, we can use **sliding windows** which apply some model to different portions of the image. However, this approach might be a bit challenging because of the large amount of areas that we would need to detect in a single image. Since the amount of pixels might be very large, we would need to beware of the amount of false-positives that are possible. For this problem, the **Viola-Jones** method is quite useful.

Viola-Jones Face Detector

The Viola-Jones method for detecting faces relies on finding **rectangle features** in images. Since a face can have eyes, nose, etc... we can measure the difference in pixel areas between similar pattern across faces. For example, if we have an idea of how different the eyes and nose should look like, we can use a model similar to Figure 2 to measure that area. If the end result is something which we might expect, then we can feel confident in saying there is a face or other feature inside.

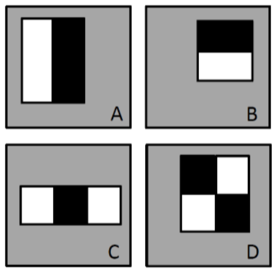

Figure 2: In Viola-Jones, there are four types of features of rectangular features. Different models might be applicable to different parts of the face.We have to take the difference of the clear and the black areas. C might be used to measure eyes and nose area; B for eyes and mouth, etc...

A "feature" in Viola-Jones corresponds to a difference in pixel values. That's why having different models with clear/black parts is useful: it can help us detect different parts in a face. Thus, for a feature value $f(x)$ given an input image vector x can be written as:

$$\text{Feature Value} = \sum(\text{Pixels in white area}) - \sum(\text{Pixels in black area})$$

The next distinguishing feature of Viola-Jones facial recognition involves the **integral image**.This aspect involves keeping a record of the sum of all preceding parts of an image. For an integral image ii, the value $ii(x, y)$ is the sum of all values *above and to the left of* x, y.

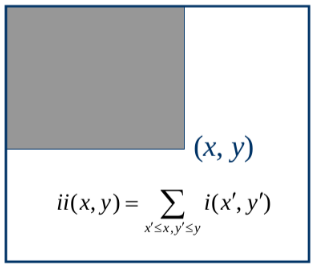

Figure 3: The value at $ii(x, y)$ is the sum of all the values that came before it.

This way of storing cumulative image values makes finding the overall value in a rectangular area quite easy. In the end, we only need a constant number of additions to calculate the feature value of any rectangular area in our image.

Boosting for face detection

So far, we've only used Viola-Jones to detect faces through the integral image. Then how do we use boosting with this approach? Similar to the definition covered before, we can use each one of the different classifiers for the rectangle areas. We can constuct a weak learner such that [5]

[5] Remember a weak learner will just work on a single recatangle area with the image.

$$h(x) = \begin{cases} 1 & p_t f_t > p_t \theta_t \\ 0 & \text{otherwise} \end{cases}$$

where p_t is the polarity, $f_t(x)$ is the value of the rectangle feature, and θ_t is some threshold. θ_t gives us our decision boundary. Thus we have over 160,000 weak learners. This face detector has complexity $O(MNK)$ for its training cycle, where M is the number of iterations, N is the training examples, and K is thnumber of rectangle features. The procedure for Viola-Jones boosting can be summarized as:

1. Evaluate the **feature rectangle** on each example.
2. select best **threshold/parity** for each rectangle feature
3. Select best **threshold/parity/rectangle-feature combination**
4. Reweigh examples

Attentional Cascade

With an **Attentional Cascade**,the idea is to run a sub window of an image through increasingly strict classifiers, and reject it as soon as one of the classifiers returns negative. With this approach, the rate of **false positives** is the product of the false positive rates of each individual classifier, which may be very little in the end.

Decision Trees

With **decision trees**, the idea is to split the space and traverse it in the same way we would a binary tree. We use a divide-and-conquer approach.[6] The idea is we have some function $f_m(x)$ at each tree node, and depending on its result we take one branch. We continue this until we reach a leaf node and can make a classification.

[6] Remember a dnd splits the problem into smaller instances,

The way a decision is made with a tree might seem quite natural to humans, since analyzing one variable at a time might seem more intuitive to make a decision..

Decision Tree Induction

To build the tree, we could first start by splitting the data into two sections, and then recursively try to split the two subsets of data themselves. We stop adding nodes when some criterion is met for the error, or as a function of the input size and model complexity. At each decison node, we can either look at **univariate** attributes or **multivariate**.

Once we start splitting the dataset, we can measure its level of **purity** by seeing if for every class C_k there are any data points in its space that do not belong to it, that belong to another class C_p, for example. We can measure the uncertainty and impurity by the following formula: [7]

[7] p_m^k gives the probability of class C_k

$$I_m = -\sum_{k=1}^{K} p_m^k \log p_m^k$$

At every decision node, we have a decision to make based on a **threshold**. In practice, decision trees might sometimes tend to overfit the training data. However, **bagging** and **random forests** might help make decision trees more robust and easier to compute.

Bagging

With **bagging**, we take a sample set of size D, and we further split it into m subsets by randomly sampling the original dataset **with replacement**. We then create separate classifiers with the different datasets and combine them in some way during the final stage to make our final classifier.

Random Forests

When using random forests, we have to introduce two new sources of randomness: **bagging** and **random vectors**. The latter refers to calculating a randomly selected subset of points and calculating the best split for each of those random vectors. How do we create a decision node? There are a few methods:

- Choose m attributes randomly and calculate the one with the biggest possible **gain**
- If M remains small, then we can choose L attributes and calculate a linear combination of those attributes that fall within $[1, -1]$.
- Compute the information gain of all the M attributes and select the top m by their information gain. Then randomly select one from within those m.

Where there are M data dimensions, and $m < M$, $L \leq M$.

For the first method, we would have to for each number in B[8] grow a decision tree based on the best attribute at each node. Then we split the dataset into two new child nodes.

[8] The number of datasets we want to split D into.