Classifier Ensembles

Constructing a good model from a given data set is one of the major tasks in machine learning (ML).

Strong classifiers are desirable, but are difficult to find.

Training many classifiers at the same time to solve the same problem, and then combining their output to improve accuracy, is known as an ensemble method.

When an ensemble, also known as a multi-classifier system, is based on learners of the same type, it is called a homogeneous ensemble.

When it is based on learners of different types, it is called a heterogeneous ensemble.

Usually, the ensemble's generalization ability is better than a single classifier's, as it can boost weak classifiers to produce better results than can a single strong classifier.

Two results published in the 1990s opened a promising new door for creating strong classifiers using ensemble methods.

The empirical study in Hansen and Salamon (1990) found that a combination of multiple classifiers produces more accurate results than the best single one, and the theoretical study in Schapire (1990) showed that weaker classifiers can be boosted to produce stronger classifiers.

There are two essential elements involved, in the design of systems that integrate multiple classifiers.

First, it is necessary to follow a plan of action to set up an ensemble of classifiers with characteristics that are sufficiently diverse.

Second, there is the need for a policy for combining the decisions, or outputs, of particular classifiers in a manner that strengthens accurate decisions and weakens erroneous classifications.

The most commonly used ensemble algorithms:

Bagging

Breiman’s bootstrap aggregating method, or “bagging” for short, was one of the first ensemble-based algorithms, and it is one of the most natural and straightforward ways of achieving a high efficiency (Breiman, 1996).

In bagging, a variety of results is produced, using bootstrapped copies of the training data, i.e. numerous subsets of data are randomly drawn with replacement from the complete training data.

A distinct classifier of the same category is modeled, using a subset of the training data.

Fusing of particular classifiers is achieved by the use of a majority vote on their selections.

Thus, for any example input, the ensemble’s decision is the class selected by the greatest number of classifiers.

Algorithm 1 contains a pseudocode for the bagging method.

Boosting

It was shown by Schapire, in 1990, that a weak learner, namely an algorithm that produces classifiers that can slightly outperform random guessing, can be transformed into a strong learner, namely an algorithm that constructs classifiers capable of correctly classifying all of the instances except for an arbitrarily small fraction (Schapire, 1990).

Boosting generates an ensemble of classifiers, as does bagging, by carrying out resampling of the data and combining decisions using a majority vote.

However, that is the extent of the similarities with bagging.

Re-sampling in boosting is carefully devised so as to supply consecutive classifiers with the most informative training data.

Essentially, **boosting generates three classifiers** as follows:

A random subset of the available training data is used for constructing the first classifier.

The most informative subset given for the first classifier is used for training the second classifier, where the most informative subset consists of training data instances, such that half of them were correctly classified by the first classifier and the other half were misclassified.

Finally, training data for the third classifier are made of instances on which the first and second classifiers were in disagreement.

A three-way majority vote is then used, to combine the decisions of the three classifiers.

How does regularization work ?

In order to find the best model, the common method in machine learning is to define a loss or cost function that describes how well the model fits the data. The goal is to find the model that minimzes this loss function.

In the case of polynomials we can define L as follows:

$$L = \sum_{m} (\sum_{i=0}^{d} (\alpha_i x_m^i) - y_m)^2$$

The idea is to penalize this loss function by adding a complexity term that would give a bigger loss for more complex models. In our case we can use the square sum of the polynomial parameters.

$$L = \sum_{m} (\sum_{i=0} (\alpha_i x_m^i) - y_m)^2 + \lambda \sum_{i=1} \alpha_i^2$$

In the visualization above, you can play around with the value of Lambda to penalize more or less the complex models.

This way, for lamda very large, models with high complexity are ruled out. And for small lambda, models with high training errors are ruled out. The optimal solution lies somewhere in the middle.

Regularization is a *technique* used in an attempt to solve the **overfitting**[1] problem in statistical models.*

First of all, I want to clarify how this problem of **overfitting** arises.

When someone wants to model a problem, let's say trying to predict the wage of someone based on his age, he will first try a linear regression model with age as an independent variable and wage as a dependent one. This model will mostly fail, since it is **too simple**.

Then, you might think: *well, I also have the age, the sex and the education of each individual in my data set. I could add these as explaining variables.*

Your model becomes **more interesting and more complex**. You measure its accuracy regarding a **loss metric** $L(X, Y)$ where X is your design matrix and Y is the observations (also denoted targets) vector (here the wages).

You find out that your result are quite good but not as perfect as you wish.

So you add more variables: location, profession of parents, social background, number of children, weight, number of books, preferred color, best meal, last holidays destination and so on and so forth.

Your model will do good but it is probably **overfitting**, i.e. it will probably have poor prediction and generalization power: it sticks too much to the data and the model has probably **learned the background noise** while being fit. This isn't of course acceptable.

So how do you solve this?

It is here where the **regularization technique** comes in handy.

You penalize your loss function by adding a multiple of an L_1 (LASSO[2]) or an L_2 (Ridge[3]) norm of your weights vector w (it is the vector of the learned parameters in your linear regression). You get the following equation:

$$L(X, Y) + \lambda N(w)$$

(N is either the L_1, L_2 or any other norm)

This will help you avoid **overfitting** and will perform, at the same time, features selection for certain regularization norms (the L_1 in the LASSO does the job).

Finally you might ask: *OK I have everything now. How can I tune in the **regularization term** λ?*

One possible answer is to use **cross-validation**: you divide your training data, you train your model for a fixed value of λ and test it on the remaining subsets and repeat this procedure while varying λ. Then you select the best λ that minimizes your loss function.

Let us understand the problem for n number of features. The problem is we are given a set of points (x1,y1),(x2,y2),(x3,y3)........(xn,yn). Our goal is to **fit** a function y=f(x) to the given data. Let us call this a **hypothesis**

$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$ for each example

where x is a vector [x1 x2 x3 x4 x5 .... xn].

We are trying the find values of θ so that function outputs a value close to given y values. Using least square error to compute our cost function, we have

Cost function = $J(\theta) = \sum_{i=1}^{m} (h_\theta(xi) - yi)^2$

A machine learning model is said to be complex when the model have so many **Thetas** that model memorizes everything in the data, "not just the signals but also the random noise, the errors, and all the slightly specific characteristics of your sample. "

So here is the thing, the cost function should be zero if our hypothesis outputs same as y. But the model might overfit the training data and might not generalize well for test data . So we want to **penalize all theta values to reduce the model complexity.** Now let's dig deep more into our cost function .

Regularization is a technique to cope with over-fitting which comes up in training a model on sample data.

Let's take a look at the simple curve fitting problem to understand regularization and over-fitting. Given a set of points $(x1, y1), (x2, y2) \ldots (xN, yN)$. Our goal is to find a model, which is a function $y = f(x)$ that fits the given data. To do this, we can use the method least-square error.

For simplicity, suppose $f(x)$ is just a first order linear function, $f(x) = Wx + b$. Our job is to figure out what W and b are. We set up an error function that looks like:

$$\sum_{i=1}^{N} (y_i \; - (Wx_i \; + \; b))^2$$

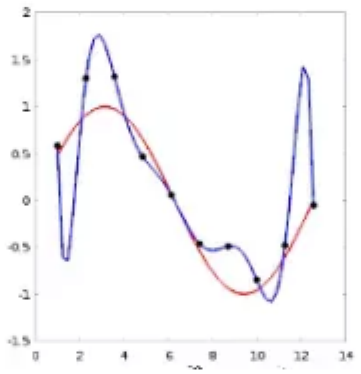
To figure out what W and b are, we need to minimize the error function above. However, in minimizing the error function, we get into a problem called over-fitting, when the model we found fits very well with the training data but fails miserably if we apply new data (i.e, get another set of data points).

To do this, we introduce a new terms into the error function, which implies that the coefficient W are also derived from a random process. The error function now looks like:

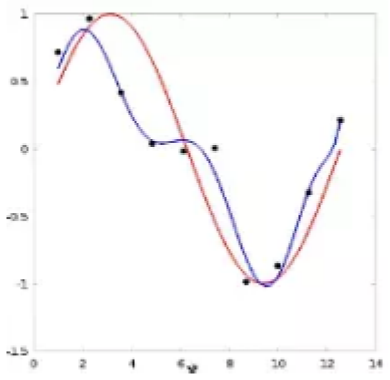
$$\sum_{i=1}^{N} (y_i \; - (Wx_i \; + \; b))^2 \; + \; \lambda(W^2)$$

The added $alpha$ is call the regularized term.

To illustrate, suppose $f(x)$ can be any order. To generate testing data, we first have a function $y = g(x)$. Next, from points belonging to $y = g(x)$, we added some noise and make those points our training data. Our goal is to derive a function $y = f(x)$ from those noisy points, that is as close to the original function $y = g(x)$ as possible. The plot below shows over-fitting, where the derived function (the blue line) fits well with the training data but does not resemble the original function.



After using regularization, the derived function looks much closer to the original function, as shown below:



Without Regularization

It can be written as in **vectorized form** $J(\theta) = (X\theta - Y)^2$.

Expanding it $(X\theta)^2 + Y^2 - 2XY\theta = J(\theta)$

Considering it as a quadratic equation in θ, it will tend to zero at its roots.

$a = X^2, b = -2XY, c = Y^2$

With Regularization

Adding regularization to our cost function, it becomes

$(X^2 + \frac{\lambda}{m})\theta^2 - 2XY\theta + Y^2 = 0$

$a = (X^2 + \frac{\lambda}{m}), b = -2XY, c = Y^2$

solving for its roots ,$= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{D}{N}$

since only constant a has changed(increased) in regularization equation, thereby decreasing D and increasing N . Hence overall value of theta decreases.

$a \uparrow \ \Rightarrow \frac{D\downarrow}{N\uparrow}$