A Lesson on Modern Classification Models

Demystifying recent advancements in ML to build you a better classification model

Jeff Da Follow

Jan 3 · 5 min read

In machine learning, **classification problems** are one of the most fundamentally *exciting* and yet *challenging* existing problems. The implications of a competent classification model are enormous—these models are leveraged for natural language processing text classification, image recognition, data prediction, reinforcement training, and a countless number of further applications.

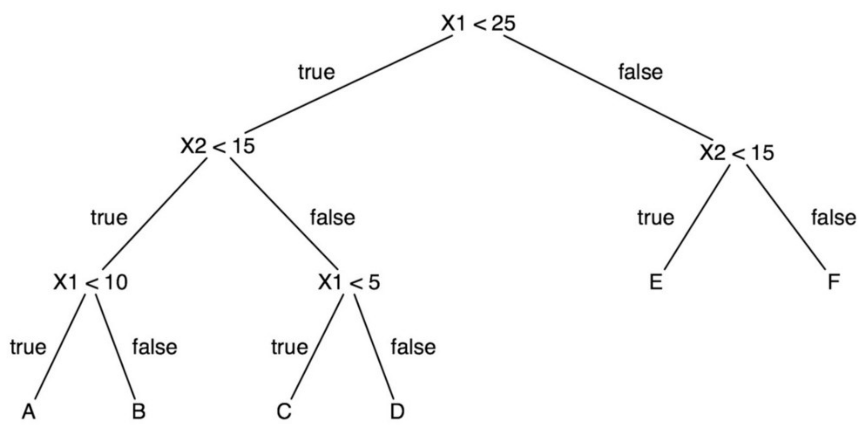
However, the present implementation of classification algorithms are terrible. During my time at Facebook, I found that the generic solution to any machine learning classification problem was to "throw a gradient descent boosting tree at it and hope for the best". But this should not be the case—research is being put into modern classification algorithms and improvements that allow *significantly* more accurate models with considerable less training data required.

Here, we explore some particularly interesting examples of modern classification algorithms. This article assumes some level of familiarity with machine learning, however, the majority of the post should still be accessible without.

Deep Neural Decision Trees

Deep neural networks have been proven powerful at processing *perceptual data*, such as images and audio. However, for tabular data, **tree-based models** are more popular for a couple reasons—one significant one being that they offer natural interpretability.

For example, consider a business that is attempting to determine why a system is failing. You would make a predictive model with several parameters—some examples are network speed, uptime, threads processing, and type of system. With a decision tree, we can additionally get a sense of *why* a system is failing.



An example of a decision tree with X1 and X2 as features. Much easier to understand than a black box neural net.

Deep neural decision trees combine the utility of a decision tree with the impact developed by neural nets. Because it is implemented with a neural network, DNDT supports out of the box GPU acceleration and min-batch learning of datasets that do not fit in memory, thanks to modern deep learning frameworks. They have thus shown to be more accurate than traditional decisions on many datasets. Furthermore, they are easy to use—an implementation involves *about 20 lines of code* in TensorFlow or PyTorch.

We can explore the concepts behind the core math of the model. First, we need a way to make the **split decisions** (i.e. how we decide which path of the tree to take). Then, we need to combine together our split decisions to build the **decision tree**.

We make the split decision via a **binning function**. A binning function takes an input and produces an index of the bins that the input belongs. In our model, each of the bins represents a feature—for example, uptime, network speed, or system type.

We bin each feature via its own *one-layer neural network* (the "deep neural" part of our model) with the following activation function.

$$\pi = softmax(\overset{\substack{\text{n bins as a vector} \\ [1,\, 2,\, \dots\, n\, -1]}}{w}x/\underset{\text{temperature factor}}{r})$$

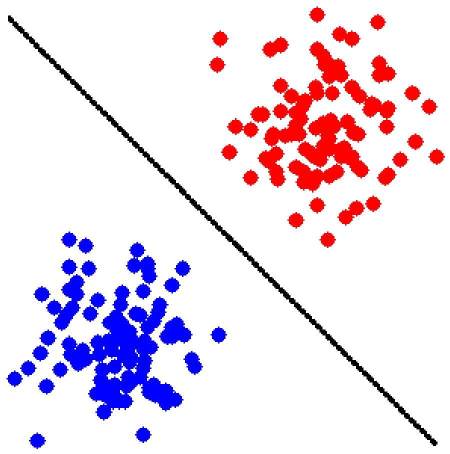
The activation function of each bin.

We then multiply our bins with each other via a Kronecker product to construct our decision tree.

$$z = f_1(x_1) \otimes f_2(x_2) \otimes \cdots \otimes f_D(x_D).$$

Each feature is binned by their own neural network. This returns us an index that represents the classification of *x* (i.e. the index of the leaf node in the tree), finishing the construction of our deep neural decision tree.

Confidence Weighted Linear Classification



Many machine learning tasks, especially in natural language processing, end up with many different features in which most are binary and infrequently operated on. This results in *data sparseness*, which demands large training sets, and very large parameter vectors.

Consider a product review classifier where the goal is to label the review as positive or negative. Many reviews might state “I liked this author”, and would thus correlate the word “liked” to a positive review. Imagine a slightly modified negative review: “I liked this author, but found this book dull”. The classifier likely would not see the world “dull” very often, and this would likely incorrectly classify this review as positive as the review uses the phrase “liked this author”, thus decreasing the rate of convergence.

We can solve this with **confidence-weighted learning**, a learning method that maintains a probabilistic measure of confidence in each parameter. Less confident parameters are updated more aggressively than more confident ones.

In confidence-weighted learning, we make sure that for each iteration, the *probability of a correct prediction* for each training instance is bigger than or equal to the *confidence for that prediction.* This can be defined formally.

$$\overset{\text{probability prediction is correct}}{P(f(x) \geq 0)} \geq \underset{\text{confidence of prediction in range [0,1]}}{n}$$

By itself, this can be appended onto any other model (you'll need to decide on how to measure confidence—some ideas are Bayesian methods such as Variational Bayes). This results in a variation of your model that trends toward solving the aforementioned problems that a sparse dataset involves.

Bayesian Principles to Determine Batch Sizing

Beyond picking a representative model, there are a set number of variables that affect the degree of accuracy our model represents. One is *batch sizing*—the amount of training data used per iteration during training. Research shows that there is not only an optimal batch size that maximises test accuracy, but that we can apply Bayesian principles to calculate as a scaling function of **learning rate** and **training data size**. The calculations itself are a multi-stepped and beyond the scope of this article, but the findings can be represented as a linear scaling function:

$$\underset{\text{batch size}}{B_{\text{optimal}}} \propto \overset{\text{learning rate}}{\epsilon}\,\underset{\text{training data size}}{N}$$

From this, we can reach two helpful conclusions.

First, as the size of our training data increases, the size of each batch iteration should increase by the same amount. Second, we should increase our batch size as our learning data increases. This allows us to determine an optimal batch size for a single experiment, and scale it to determine batch sizes for any number of varied experiments, optimizing test accuracy across the board.

Conclusion

This is just a taste of the breadth (and depth) of classification problems in modern machine learning research—everything in this article was published in the last few months. Hopefully, this shows you a little bit of the exciting complexity that one might consider when choosing their classification models in the future!