Computational Statistics Project

Neel Adhiraj Kumar

12/12/2016

Boosting

In 1988, Michael Kearns posed the Hypothesis Boosting Problem.

> "Informally, this problem asks whether an efficient learning algorithm... that outputs an hypothesis whose performance is only slightly better than random guessing implies the existence of an efficient algorithm that outputs an hypothesis of arbitrary accuracy... from the theoretical standpoint, we are interested more generally in the question of whether there is a discrete hierarchy of achievable accuracy... from a practical standpoint, the collapse of such a proposed hierarchy may yield an efficient algorithm for converting relatively poor hypotheses into very good hypotheses."[7]

This gave birth to the idea of boosting - combining weak (better than random) learners into a strong learner.

> "The idea is to used the weak learning method several times to get a succession of hypotheses, each one refocused on the examples that the previous ones found difficult and misclassified."[9]

AdaBoost

The first implementation of this idea was by Yoav Freund and Robert Schapire, in a meta-algorithm called Adaptive Boosting, or AdaBoost for short. In each iteration, AdaBoost weights harder to classify observations more heavily, and hence new learners focus more on those cases than the ones the model already got right, until we find a learner that could deal with them too. Predictions were made by vote of the weak learners individual predictions, weighted by individual accuracy.

This was a major breakthrough, since it showed that a polynomial time learner generating hypotheses with error slightly less than .5 could be converted to a polynomial time learner generating hypotheses with an arbitrarily small error [8]. Adaboost can be used in conjunction with many types of learning algorithms to improve their performance, though typically it's used with decision trees with a single split, called decision stumps for their simplicity.

Pseudocode for the Adaboost meta-algorithm:

> We are given $(x_1, y_1), ..., (x_m, y_m)$ where $x_i \in X$, some domain, and the labels $y_i \in \{-1, 1\}$.
>
> We initialise our distribution, say with $D_1(i) = \frac{1}{m}$ for $i = 1, ..., m$.
>
> On each round $t = 1, ..., T$ a distribution D_t is computed over the m training examples, and a given weak learner or weak learning algorithm (like decision trees) is applied to find a weak hypothesis $h_t : X \to \{-1, 1\}$, where the aim of the weak learner is to find a weak hypothesis with low weighted error ϵ_t relative to D_t.

$$\epsilon_t = Pr_{i \approx D_t}[h_t(x_i) \neq y_i]$$

The final or combined hypothesis H computes the sign of a weighted combination of weak hypotheses. We can weight the hypotheses for instance, with

$$\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$$

which is always positive since $\epsilon_t < 0.5$

Then we get,

$$H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

This is equivalent to saying that H is computed as a weighted majority vote of the weak hypotheses h_t where each is assigned weight α_t.[8]

AdaBoost gives generalisation errors significantly lower than those produced by bagging, the first well known combination algorithm [1]. Unlike neural networks and SVMs, AdaBoost during training selects only those features that actually improve the model's prediction, reducing dimensionality and potentially improving execution time as irrelevant features are not computed. In 2003, Fruend and Schapire won the Godel Prize for their work.

Gradient Boosting

Freidman recast boosting as a numerical optimisation problem, where like gradient descent, the algorithm's objective was to minimise the loss function by taking a 'step' in the steepest-descent direction at each iteration, by adding weak learners. This is known as gradient boosting.[6]

This is a stage-wise additive model, because only one new learner is added at a time, and previous learners are left unchanged (as opposed to a stepwise learner that readjusts previous learners when new ones are added).

This generalisation allowed boosting to use arbitrary (differentiable) loss functions, extending it's domain from binary classification to regression, multi-class classification, and more.

Implementation:

The gradient boosting algorithm requires 3 components:

1. **A (differentiable) loss function**.

   An advantage of gradient boosting is that we can change the loss function without having to change the entire algorithm, since the algorithm works for any differentiable loss function. Of course, we can add regularisation to our loss function.[10]

$$obj = \sum_{i=1}^{n} l(y_i, \hat{y_i}) + \sum_{k=1}^{K} \Omega(f_k)$$

   The actual function depends on the problem being solved. For instance,

   For **regression** we can use ordinary least squares. If our model is $F(x)$,

$$L(y, F(x)) = \frac{(y - F(x))^2}{2}$$

We want to minimise $J = \sum_i L(y_i, F(x_i))$ by adjusting $F(x_1), ..., F(x_n)$. Treating $F(x_i)$ as parameters and taking derivatives,

$$\frac{\delta J}{\delta F(x_i)} = \frac{\delta \sum_j L(y_j, F(x_j))}{\delta F(x_i)} = \frac{\delta L(y_i, F(x_i))}{\delta F(x_i)} = F(x_i) - y_i$$

Thus if we use ordinary least squares, the negative of the gradient is just the residual $y_i - F(x_i)$. Hence we start with an initial model, find the residual of that model, fit a weak learner (like a decision tree) to that residual, update the model, and repeat. This makes intuitive sense, but the notion of gradient is more general than just the residual (which is specific to least squares).

2. **A weak learner**

   Usually, gradient boosting algorithms use decision trees. Specifically, we want regression trees, whose real valued outputs we can add, allowing more models to be added and "correct" the error in the prediction.

   We construct the trees by choosing the best split points each time, based on purity scores like Gini or Entropy or to minimize the loss function, in a "greedy algorithm".

   Initially algorithms like AdaBoost used very short decision trees. We can use larger trees, usually with 4-to-8 levels. We constrain the learners with a limit on the number of layers, nodes, splits, or leaf nodes, to make the algorithm computationally efficient.

3. **An additive model**

   We add trees one at a time, and don't change existing trees. A gradient descent procedure is used to minimize the loss when adding trees.

   Traditionally, we use gradient descent to minimise a set of parameters, such as the coefficients in a regression equation. After calculating error or loss, the weights are updated to minimize that error. In this case instead of parameters we have weak learners, specifically decision trees. After calculating the loss, at each step we use gradient descent to add a tree to the model that reduces the loss (i.e. follow the gradient). This approach is called functional gradient descent.

   The output for the new tree is then added to the output of the existing sequence of trees in an effort to correct or improve the final output of the model. A fixed number of trees are added or training stops once loss reaches an acceptable level or no longer improves on an external validation dataset.

XGBoost

XGBoost, or Extreme Gradient Boosting, is an implementation of gradient boosted decision trees focused on optimising speed and model performance. It is highly effective, and by far the most popular algorithm today for applied machine learning and Kaggle competitions for structured data. It is generally over 10 times faster than *gbm*. We can download and install it and use it through a variety of interfaces, including R, which makes implementing gradient boosting very straightforward.[4]

It supports many objective functions, including for regression, classification and ranking. Users can also define their own objectives. We can use Gradient Boosting, Stochastic Gradient Boosting with sub-sampling at the row, column and column per split levels, and Regularized Gradient Boosting with both L1 and L2 regularization. Below we present an example using xgboost [5][10].

Agaricus is from the Mushroom data set, UCI Machine Learning Respository. It is a matrix object of 6513 rows and 127 variables, one of which is a label. The labels are inedible (unknown edibility or definitely poisonous) and definitely edible. The data set describes hypothetical samples of 23 species of gilled mushrooms in the Agaricus and Lepiota Family, with the various attributes encoded in the 126 variables. Hence this is a binary classification problem. The dataset also has many missing values.

```
set.seed(1)
require(xgboost)
```

```
## Loading required package: xgboost
```

```
data(agaricus.train, package='xgboost') #loading data
data(agaricus.test, package='xgboost')
train <- agaricus.train
test <- agaricus.test
dim(train$data)
```

```
## [1] 6513  126
```

```
dim(test$data)
```

```
## [1] 1611  126
```

```
#using built in data types of package for best performance
dtrain <- xgb.DMatrix(train$data, label=train$label)
dtest <- xgb.DMatrix(test$data, label=test$label)
watchlist <- list(train=dtrain, test=dtest)
#each tree is only 2 levels with learning rate of each tree=1
param <- list(max.depth = 2, eta = 1)
bst <- xgb.train(data=dtrain, params = param, nround=10, watchlist=watchlist, objective = "binary:logis
```

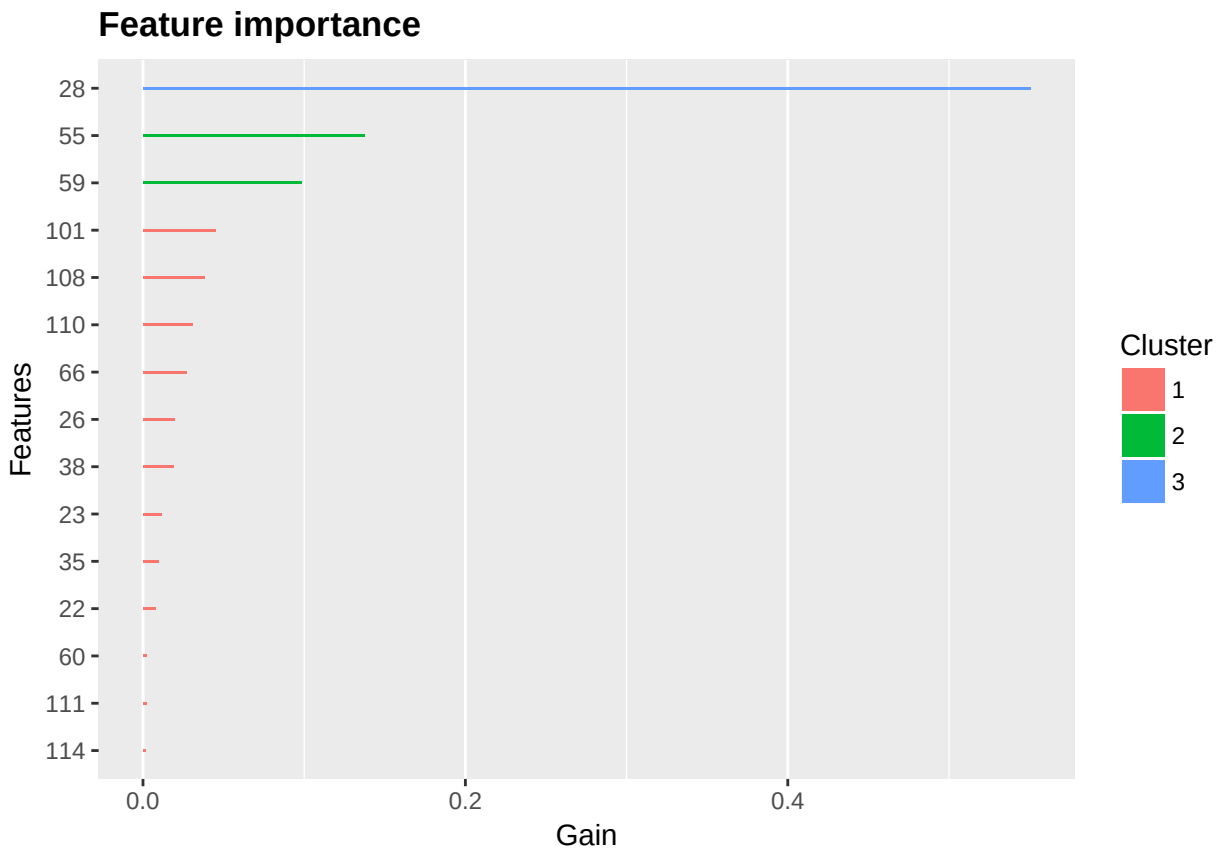
```
## [0]  train-error:0.046522    train-logloss:0.233357  test-error:0.042831 test-logloss:0.226688
## [1]  train-error:0.022263    train-logloss:0.136649  test-error:0.021726 test-logloss:0.137877
## [2]  train-error:0.007063    train-logloss:0.082530  test-error:0.006207 test-logloss:0.080462
## [3]  train-error:0.015200    train-logloss:0.056476  test-error:0.018001 test-logloss:0.058329
## [4]  train-error:0.007063    train-logloss:0.041513  test-error:0.006207 test-logloss:0.038287
## [5]  train-error:0.001228    train-logloss:0.029607  test-error:0.000000 test-logloss:0.026631
## [6]  train-error:0.001228    train-logloss:0.019191  test-error:0.000000 test-logloss:0.013875
## [7]  train-error:0.001228    train-logloss:0.013320  test-error:0.000000 test-logloss:0.010198
## [8]  train-error:0.001228    train-logloss:0.011130  test-error:0.000000 test-logloss:0.008483
## [9]  train-error:0.000000    train-logloss:0.006634  test-error:0.000000 test-logloss:0.006920
```

```
pred <- predict(bst, test$data) #we predict probabilities of classification
err <- mean((abs(pred-test$label))) #error in our probability
print(c("Average error is: ",err))
```

```
## [1] "Average error is: "  "0.00664957362573899"
```

We see that we can get a "perfect" model pretty quickly, which despite overfitting the training data, gives great results for the test data. We are free to pick any of the models provided, all of which give a very low error. We can also see the importance of various features from the model.

```
importance_matrix <- xgb.importance(model = bst)
xgb.plot.importance(importance_matrix)
```



xgboost provides a variety of other features to use. We can use linear learners instead of trees (with regularisation), scale the contribution of each tree by a factor to prevent overfitting, set a minimum loss reduction required for splits, set a maximum depth for trees, set minimum weights for a child, grow trees from a subsample of the data, provide a customised objective function, and several others.

Thus, we have boosting, an incredibly powerful idea to go from weak models (barely better than random) to an arbitrarily strong one. The gradient boosting algorithm provides a powerful way to apply this idea to a broad class of problems, and the xgboost package makes it incredibly easy to implement it to our specifications, while ensuring computational resources are maximised. Given the intuitive appeal of the idea, the speed and performance of the model, and how easy and flexible it is to implement, there is really no wonder it is a favourite for data analysis.

References

1. Breiman, Leo. "Prediction Games and Arcing Algorithms." Neural Computation 11.7 (1999): 1493-517. Web. 15 Dec. 2016.

2. Brownlee, Jason. "A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning - Machine Learning Mastery." Machine Learning Mastery. N.p., 9 Sept. 2016. Web. 16 Dec. 2016..

3. Brownlee, Jason. “A Gentle Introduction to XGBoost for Applied Machine Learning - Machine Learning Mastery.” Machine Learning Mastery. N.p., 17 Aug. 2016. Web. 16 Dec. 2016.

4. Chen, Tianqi, and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System.” Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD ’16 (2016): n. pag. Web.

5. Chen, Tianqi, He, Tong, and Benesty, Michael (2016). xgboost: Extreme Gradient Boosting. R package version 0.4-4. https://CRAN.R-project.org/package=xgboost

6. Friedman, Jerome H. “Greedy Function Approximation: A Gradient Boosting Machine.” IMS 1999 Reitz Lecture. Stanford University, Stanford. Lecture.

7. Kearns, Michael. “Thoughts on Hypothesis Boosting.” N.d. TS. University of Pennsylvania, n.p.

8. Schapire, Robert E. “Explaining AdaBoost.” Empirical Inference (2013): 37-52. Web.

9. Valiant, Leslie. Probably approximately correct: nature’s algorithms for learning and prospering in a complex world. pg 152. New York: Basic , 2013. Print.

10. “XGBoost R Tutorial.” XGBoost R Tutorial - xgboost 0.6 documentation. N.p., n.d. Web. 16 Dec. 2016. http://xgboost.readthedocs.io/en/latest/R-package/xgboostPresentation.html.