Gradient boosting

Yue Jing yj1142

General definition/description of the algorithm:

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

Like other boosting methods, gradient boosting combines weak "learners" into a single strong learner in an iterative fashion. It is easiest to explain in the least-squares regression setting, where the goal is to "teach" a model F to predict values in the form $\hat{y} = F(x)$ by minimizing the mean squared error $(\hat{y} - y)^2$, averaged over some training set of actual values of the output variable y.

At each stage m, $1 \leq m \leq M$ of gradient boosting, it may be assumed that there is some imperfect model Fm (at the outset, a very weak model that just predicts the mean y in the training set could be used). The gradient boosting algorithm improves on Fm by constructing a new model that adds an estimator h to provide a better model: $F_{m+1}(x) = F_m(x) + h(x)$. To find h, the gradient boosting solution starts with the observation that a perfect h would imply

$$F_{m+1}(x) = F_m(x) + h(x) = y$$

or, equivalently,

$$h(x) = y - F_m(x).$$

Therefore, gradient boosting will fit h to the residual $y - F_m(x)$. Like in other boosting variants, each F_{m+1} learns to correct its predecessor F_m. A generalization of this idea to loss functions other than squared error - and to classification and ranking problems - follows from the observation that residuals $y - F(x)$ for a given model are the negative gradients (with respect to $F(x)$) of the squared error loss function $\frac{1}{2}(y - F(x))^2$. So, gradient boosting is a gradient descent algorithm; and generalizing it entails "plugging in" a different loss and its gradient.

In many supervised learning problems one has an output variable y and a vector of input variables x connected via a joint probability distribution $P(x, y)$. Using a training set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ of known values of x and corresponding values of y, the goal

is to find an approximation $\hat{F}(x)$ to a function $F_{(x)}$ that minimizes the expected value of some specified loss function $L(y, F(x))$:

$$\hat{F} = \arg\min_{F} \mathbb{E}_{x,y}[L(y, F(x))]$$

The gradient boosting method assumes a real-valued y and seeks an approximation $\hat{F}(x)$ in the form of a weighted sum of functions $h_i(x)$ from some class $\mathcal{H}$, called base (or weak) learners:

$$F(x) = \sum_{i=1}^{M} \gamma_i h_i(x) + \text{const.}$$

In accordance with the empirical risk minimization principle, the method tries to find an approximation $\hat{F}(x)$ that minimizes the average value of the loss function on the training set. It does so by starting with a model, consisting of a constant function $F_0(x)$, and incrementally expanding it in a greedy fashion:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma),$$

$$F_m(x) = F_{m-1}(x) + \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h_m(x_i)),$$

where $h \in \mathcal{H}$ is a base learner function.

Unfortunately, choosing the best function h at each step for an arbitrary loss function L is a computationally infeasible optimization problem in general. Therefore, we will restrict to a simplification.

The idea is to apply a steepest descent step to this minimization problem. If we considered the continuous case, i.e. $\mathcal{H}$ the set of arbitrary differentiable functions on $\mathbb{R}$, we would update the model in accordance with the following equations

$$F_m(x) = F_{m-1}(x) - \gamma_m \sum_{i=1}^{n} \nabla_{F_{m-1}} L(y_i, F_{m-1}(x_i)),$$

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) - \gamma \nabla_{F_{m-1}} L(y_i, F_{m-1}(x_i))\right),$$

where the derivatives are taken with respect to the functions F_i for $i \in \{1, .., m\}$. In the discrete case however, i.e. the set $\mathcal{H}$ is finite, we will choose the candidate function h closest to the gradient of L for which the coefficient γ may then be calculated with the aid of line search the above equations. Note that this approach is a heuristic and will therefore not yield an exact solution to the given problem, yet a satisfactory approximation.

In pseudocode, the generic gradient boosting method is:

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For m = 1 to M:
    1. Compute so-called *pseudo-residuals*:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

    2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

    4. Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

3. Output $F_M(x)$.

Mathematical Formula the algorithm relies on:

Gradient tree boosting

Gradient boosting is typically used with decision trees of a fixed size as base learners. For this special case Friedman proposes a modification to gradient boosting method which improves the quality of fit of each base learner.

Generic gradient boosting at the m-th step would fit a decision tree $h_m(x)$ to pseudo-residuals. Let Jm be the number of its leaves. The tree partitions the input space into Jm disjoint regions $R_{1m}, \ldots, R_{J_m m}$ and predicts a constant value in each region. Using the indicator notation, the output of $h_m(x)$ for input x can be written as the sum:

$$h_m(x) = \sum_{j=1}^{J_m} b_{jm} \mathbf{1}_{R_{jm}}(x),$$

where b_{jm} is the value predicted in the region R_{jm}

Then the coefficients b_{jm} are multiplied by some value γ_m, chosen using line search so as to minimize the loss function, and the model is updated as follows:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x), \quad \gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)).$$

Friedman proposes to modify this algorithm so that it chooses a separate optimal value γ_{jm} for each of the tree's regions, instead of a single γ_m for the whole tree. He calls the modified algorithm "TreeBoost". The coefficients b_{jm} from the tree-fitting procedure can be then simply discarded and the model update rule becomes:

$$F_m(x) = F_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} \mathbf{1}_{R_{jm}}(x), \quad \gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma).$$

Size of trees

J, the number of terminal nodes in trees, is the method's parameter which can be adjusted for a data set at hand. It controls the maximum allowed level of interaction between variables in the model. With $J = 2$ (decision stumps), no interaction between variables is allowed. With $J = 3$ the model may include effects of the interaction between up to two variables, and so on.

Hastie et al. comment that typically $4 \leq J \leq 8$ work well for boosting and results are fairly insensitive to the choice of J in this range, $J = 2$ is insufficient for many applications, and $J > 10$ is unlikely to be required.

Generative/Discriminative algorithm and why so:

In probability and statistics, a **generative** model is a model for generating all values for a phenomenon, both those that can be observed in the world and "target" variables that can only be computed from those observed.
By contrast, **discriminative** models provide a model only for the target variable(s), generating them by analyzing the observed variables. In simple terms, discriminative models infer outputs based on inputs, while generative models generate both inputs and outputs, typically given some hidden parameters.
Decision trees are discriminative because it learns explicit boundaries between classes. DTs learn the decision boundary by recursively partitioning the space in a manner that maximizes the information gain (or another criterion). Discriminative models do not offer such clear representations of relations between features and classes in the dataset. Instead of using resources to fully model each class, they focus on richly modeling the boundary between classes. Given the same amount of capacity (say, bits in a computer program executing the model), a discriminative model thus may yield more complex representations of this boundary than a generative model.
Gradient tree boosting is a kind of decision tree. Therefore, based on the definition of discriminative and decision tree, we can say that Gradient boosting is discriminative algorithm.

Any assumptions the algorithm relies on:

Like decision trees, boosting makes no assumptions about the distribution of the data. The key parameters controlling model complexity for gradient boosted tree models are, n_estimators which sets the number of small decisions trees the week learns to use in the ensemble, and the learning rate. Typically, these two parameters are tuned together. Since making the learning rates smaller, will require more trees to maintain model complexity. Unlike random forest, increasing an n_estimators can lead to overfeeding. So typically, the n_estimators setting is chosen to best exploit the speed and memory capabilities of the system during the training. And other parameters like the learning rate are then adjusted, given that fixed an n_estimators setting. The max_depth parameter can also have an effect of model complexity, but controlling the depth, and has a complexity of the individual trees.
The gradient boosting method assumes, that each tree is a weak learner, and so the max depth parameter is usually quite small, on the order of three to five, for most applications.

Restrictions on the independent variable (can they be categorical and numerical, scale, variance), Restrictions on the dependent variable (categorical or numerical, scale, variance):

For the Gradient Boosting Classification operator, there are not restrictions on the independent variable. They can be both categorical, numerical and etc. However, the dependent column must be a **binary** categorical or continuous numerical value that has only **0** or **1** as its values. Multi-class classification is currently not supported. And we need to be sure that the values chosen for the Independent and the Dependent variable are different. If the user chooses a field as a dependent variable that is also selected in the Independent variable list, an error will occur.

Dataset you ran with:

In the project, we used the titanic data set from
http://christianherta.de/lehre/dataScience/machineLearning/data/titanic-train.csv

```
library(rpart)
titanic <- read.csv("titanic-train.csv",header=T,stringsAsFactors = F)
data <- titanic[complete.cases(titanic),]
set.seed(43)
test_id <- sample(714, 200, replace=FALSE)
titanic_testdata <- data[test_id,]
titanic_traindata <- data[-test_id,]
write.csv(titanic_testdata, "/Users/yuejing/Downloads/titanic_test.csv")
write.csv(titanic_traindata, "/Users/yuejing/Downloads/titanic_train.csv")
```

In order to compare the accuracy, confusion matrix and AUC between R and Spark (Java). I used seed(43) function in R to separate the titanic-train.csv file and save the result to train data (titanic_train.csv) and test data (titanic_test.csv). After de-duplicate the data, I put 200 of them in test data, and the rest of them (714) in train data.

R script to run Gradient Boosting:

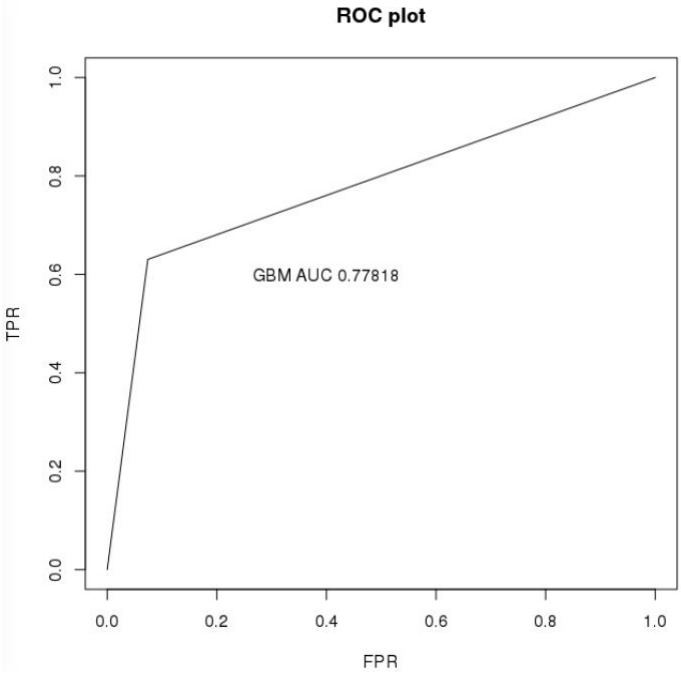
```
library(caret)
library(rpart)
library(gbm)

titanic_train <- read.csv("titanic_train.csv",header=T,stringsAsFactors = F)
titanic_test <- read.csv("titanic_test.csv",header=T,stringsAsFactors = F)

traindata <- titanic_train[,c(3,4,6,7,11)]
trainsex <- ifelse(traindata$Sex=="male", 1, 0)
traindata[,"Sex"] <- trainsex
testdata <- titanic_test[,c(3,4,6,7,11)]
testsex <- ifelse(testdata$Sex=="male", 1, 0)
testdata[,"Sex"] <- testsex

gbm_data <- gbm(Survived ~ ., cv.folds=11,n.cores=2,interaction.depth=5,shrinkage =
0.0005,distribution="adaboost",data=traindata,n.trees=10000)
gbm_pred <- predict(gbm_data,testdata[,2:5], type="response")
sol <- as.numeric(gbm_pred > 0.5)
result <- confusionMatrix(sol, testdata$Survived)
print(result)

#compute AUC
require(ROCR)
gbm_auc <- prediction(gbm_pred, testdata$Survived)
gbm_prf <- performance(gbm_auc, measure="tpr", x.measure = "fpr")
gbm_perf_AUC = performance(gbm_auc, "auc")
gbm_AUC_y = gbm_perf_AUC@y.values[[1]]
print(gbm_AUC_y)
#gbm_slot_fp <- slot(gbm_auc, "fp")
#gbm_slot_tp <- slot(gbm_auc, "tp")
#gbm_fpr <- unlist(gbm_slot_fp)/unlist(slot(gbm_auc, "n.neg"))
#gbm_tpr <- unlist(gbm_slot_tp)/unlist(slot(gbm_auc, "n.pos"))
plot(gbm_prf, main = "ROC plot", xlab = "FPR", ylab = "TPR")
text(0.4,0.6,paste("GBM AUC",format(gbm_AUC_y,digits = 5, scientific = FALSE)))
```

R Accuracy and Confusion Matrix

```
> print(result)
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 100  34
         1   8  58

               Accuracy : 0.79
                 95% CI : (0.7269, 0.8443)
    No Information Rate : 0.54
    P-Value [Acc > NIR] : 1.563e-13

                  Kappa : 0.5683
 Mcnemar's Test P-Value : 0.0001145

            Sensitivity : 0.9259
            Specificity : 0.6304
         Pos Pred Value : 0.7463
         Neg Pred Value : 0.8788
             Prevalence : 0.5400
         Detection Rate : 0.5000
   Detection Prevalence : 0.6700
      Balanced Accuracy : 0.7782

       'Positive' Class : 0
```

R AUC value:

```
[1] 0.7781804
```



Java code to run Gradient Boosting in Spark ML:

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics;
import org.apache.spark.mllib.linalg.DenseVector;
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.log4j.LogManager;
import org.apache.spark.SparkConf;
import java.util.HashMap;
import java.util.Map;
import org.apache.spark.mllib.tree.GradientBoostedTrees;
import org.apache.spark.mllib.tree.configuration.BoostingStrategy;
import org.apache.spark.mllib.tree.model.GradientBoostedTreesModel;
import scala.Tuple2;

public class TitanicProblemSpark {

   public static final String COMMA_DELIMITER = ",(?=([^\"]*\"[^\"]*\")*[^\"]*$)";

   public static void main(String[] args) {
      LogManager.getLogger("org").setLevel(org.apache.log4j.Level.OFF);
      SparkConf sparkConf = new SparkConf().setMaster("local[*]").setAppName("Titanic Spark");
      JavaSparkContext javaSparkContext = new JavaSparkContext(sparkConf);

      JavaRDD<String> datatrain = javaSparkContext.textFile("titanic_train.csv").filter(line
-> !"".equals(line.split(COMMA_DELIMITER)[5]));
      JavaRDD<String> datatest = javaSparkContext.textFile("titanic_test.csv").filter(line
-> !"".equals(line.split(COMMA_DELIMITER)[5]));

      JavaRDD<LabeledPoint> trainingData = datatrain.map(line -> {
         String[] params = line.split(COMMA_DELIMITER);
         double label = Double.valueOf(params[2]);
         double[] vector = new double[4];
         vector[0] = Double.valueOf(params[3]);
         vector[1] = "male".equals(params[5]) ? 1d : 0d;
         vector[2] = Double.valueOf(params[6]);
         vector[3] = Double.valueOf(params[10]);
         return new LabeledPoint(label, new DenseVector(vector));
      });
```

```
JavaRDD<LabeledPoint> testData = datatest.map(line -> {
        String[] params = line.split(COMMA_DELIMITER);
        double label = Double.valueOf(params[2]);
        double[] vector = new double[4];
        vector[0] = Double.valueOf(params[3]);
        vector[1] = "male".equals(params[5]) ? 1d : 0d;
        vector[2] = Double.valueOf(params[6]);
        vector[3] = Double.valueOf(params[10]);
        return new LabeledPoint(label, new DenseVector(vector));
      });

// Train a GradientBoostedTrees model.
// The defaultParams for Classification use LogLoss by default.
BoostingStrategy boostingStrategy = BoostingStrategy.defaultParams("Classification");
boostingStrategy.setNumIterations(3); // Note: Use more iterations in practice.
boostingStrategy.getTreeStrategy().setNumClasses(2);
boostingStrategy.getTreeStrategy().setMaxDepth(5);
// Empty categoricalFeaturesInfo indicates all features are continuous.
Map<Integer, Integer> categoricalFeaturesInfo = new HashMap<>();
boostingStrategy.treeStrategy().setCategoricalFeaturesInfo(categoricalFeaturesInfo);

final GradientBoostedTreesModel model = GradientBoostedTrees.train(trainingData,
boostingStrategy);

// Evaluate model on test instances and compute test error
JavaPairRDD<Object, Object> predictionAndLabel =
      testData.mapToPair( p ->
            (new Tuple2<>(model.predict(p.features()), p.label()))
      );
BinaryClassificationMetrics metrics =
      new BinaryClassificationMetrics(predictionAndLabel.rdd());
Double testErr =
      1.0 * predictionAndLabel.filter(pl -> (!pl._1().equals(pl._2()))
      ).count() / testData.count();
Double accuracy = 1 - testErr;
System.out.println("Accuracy: " + accuracy);
```

Spark Accuracy:

```
Accuracy: 0.785
```

Compute the confusion matrix and the AUC number:

```
//Confusion Matrix and Statistics
long[][] matrix = new long[2][2];
long truePositive = predictionAndLabel.filter(pl -> (pl._1()).equals(pl._2()) &&
pl._1().equals(1d)).count();
matrix[1][1] = truePositive;
long trueNegative = predictionAndLabel.filter(pl -> (pl._1()).equals(pl._2())
&& !pl._1().equals(1d)).count();
matrix[0][0] = trueNegative;
long falsePositive = predictionAndLabel.filter(pl -> !(pl._1()).equals(pl._2()) &&
pl._1().equals(1d)).count();
matrix[1][0] = falsePositive;
long falseNegative = predictionAndLabel.filter(pl -> !(pl._1()).equals(pl._2())
&& !pl._1().equals(1d)).count();
matrix[0][1] = falseNegative;
System.out.println("Confustion Matrix: " );
for (int i = 0; i < matrix.length; i++) {
   for (int j = 0; j < matrix[i].length; j++) {
      System.out.print(matrix[i][j] + " ");
   }
   System.out.println();
}

//AUC number
System.out.println("Area under ROC = " + metrics.areaUnderROC());
```

Spark Confusion Matrix and AUC value:

```
Confustion Matrix:
101 36
7 56
Area under ROC = 0.7719404186795491
```

Compare results from R and results from Java:

Accuracy: R: 0.79 Spark (Java): 0.785

The Confusion matrix:

R:

	Reference	0	1
Prediction	0	100	34
	1	8	58

Spark (Java):

	Reference	0	1
Prediction	0	101	36
	1	7	56

AUC value: R: 0.7781804 Spark (Java): 0.77194

We can find that accuracy, confusion matrix and AUC value from R and Spark (Java) are similar. Therefore, we can say that using these two methods we can get same analysis solution.

Summary:

In this project, I learnt to use both R and Spark (Java) to analysis the data using Machine Learning algorithm (Gradient Boosting Machine). And using the accuracy, confusion matrix and AUC value to compare the difference between different methods.

To finish this project, first I spent several hours to understand Gradient Boosting Machine. Then I spent about 10 hours on R. Around 3 hours to do the research and rest of time to write the code. Next, I spent more than one day (24 hours) to do the research on Spark (Java) and write the code. Finally, I found some source and wrote the report.

According to the time I spent on this project. Spark (Java) is harder than R, and both of them are very useful. It’s helpful for me to do this project, I have deeper understanding of machine learning and big data analysis.

Files for project2 - TitanicProblemSpark folder, TitanicProblemSpark-1.0-SNAPSHOT.jar, titanic-train.csv, titanic_train.csv, titanic_test.csv, project2_yj1142.R, Project_Report2.pdf

References:

1. Ensembles - RDD-based API
https://spark.apache.org/docs/2.1.0/mllib-ensembles.html#gradient-boosted-trees-gbts

2. Gradient boosting
https://en.wikipedia.org/wiki/Gradient_boosting

3. A Kaggle Master Explains Gradient Boosting
http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/

4. Package ‘gbm’
https://cran.r-project.org/web/packages/gbm/gbm.pdf

5. Evaluating classifierperformance ml-cs6923
https://www.slideshare.net/rk2153/evaluating-classifierperformance-mlcs6923

6. Gradient Boosting Classification
https://alpine.atlassian.net/wiki/spaces/V6/pages/101155133/Gradient+Boosting+Classification