CMPSC 448

HW4

Problem 2

* XGBoost(Boosted Decision Trees)

1. Description

Boosting basic ideas:

Boosting refers to a general and provably effective method of producing a very accurate classifier by combining by combining rough and moderately inaccurate rules of thumb. Instead of learning a single (weak) classifier, learn many weak classifiers that are good at different parts of the input space.

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.

Background:

Most serious ML practitioners are already familiar, or at least have heard of XGBoost (short for EXtreme Gradient Boosting). XGBoost is a gradient boosting library that came to fame after winning the Kaggle Higgs Machine Learning Challenge in 2014. This library became popular for several reasons:

It implemented very effective regularization hyperparameters on the underlying trees.

It was built to be highly scalable, utilizing distributed computing to process extremely large data sets.

XGBoost still does quite well in modern competitions, and the XGBoost community has done a great job of maintaining the packages and adding new features.

1. Training methodology
2. Get the default parameter from official document
3. Set the default as the constant, each time using 5-fold cross evaluation to figure out one optimal parameter based on default value; In other word, set one parameter as a variable and the rest of parameter as constants. The step is to narrow the range of input parameters.
4. repeat #2 k times to get approximate optimal parameters of range (k is the number of parameters to fine turning)
5. combine every improved parameter together then evaluate the model and output the result
6. random input parameters in the decision range to see whether get a better performance

(please see the attached file for more details)

1. Hyperparameters & Error rates

|  |  |  |
| --- | --- | --- |
|  | default | best result |
| Training error rates |  | 0.144732801 |
| cross-validation error rates |  | 0.153178476 |
| test error rates |  | 0.147533935 |
| n\_estimators | 100 | 70 |
| max\_depth | 6 | 5 |
| lambda | 1 | 1 |
| learning\_rate | 0.3 | 0.1 |
| missing | nan | nan |
| objective | binary:logistic | reg:logistic |

* SVM (Support Vector Machines with Gaussian Kernel)

1. Description

SVM basic ideas:

Support vector machines so called as SVM is a supervised learning algorithm which can be used for classification and regression problems as support vector classification (SVC) and support vector regression (SVR).

We basically consider that the data is linearly separable and this might not be the case in real life scenario. In some case, linear separability is impossible in most real datasets. We need input data to a high-dimensional feature space (hopefully data becomes separable in mapped spaces) and then run linear SVM using the kernel trick to make the calculation efficient.

The kernel trick:

Using kernel, we can obtain non-linear methods from linear methods as follows:

1. Select an inner product kernel k
2. Formulate your linear method such that feature vector only appear via inner product form <xi, xj>
3. Replace <xi,xj> with k(xi,xj) through your algorithm
4. Training methodology

Get the default parameter from official document

* + 1. This time our parameters’ dimensional is much small than before. For example, kernel will only have four potential choices ['linear', 'poly', 'rbf', 'sigmoid'] and similarly, gamma only have two choices ['scale','auto'].

That mean we can use brute force method to try out every possible solutions on those parameters above

(please see the attached file for more details)

1. Hyperparameters & Error rates

|  |  |  |
| --- | --- | --- |
|  | default | best result |
| Training error rates |  | 0.131718366 |
| cross-validation error rates |  | 0.153102194 |
| test error rates |  | 0.152570481 |
| kernel\_type | rbf | poly |
| gamma | scale | scale |
| C | 1 | 1 |

* Random Forests

1. Description

Background:

Bagging seems to work especially well for high-variance, low-bias procedures, such as trees but the trees produced by different Bootstrap samples can be very similar.

To fix this problem, random forests provide an improvement over bagged trees by way of a small tweak to de-correlate the models

As in bagging, we build a number of decision trees on bootstrapped training samples.

But when building these decision trees, each time a split in a tree is considered, a random sample of p < d features (predictors) is chosen as split candidates from the full set of all d features. The split is allowed to use only one of those P features. A fresh sample of features is taken at each split, and typically we choose p = d ^ 0.5

Training methodology

1. Get the default parameter from official document. I will use the similar methodology in the XGBoost. I could use a huge loop to loop every possible variable but my PC’s computering power is not good enough
2. Set the default as the constant, each time using 5-fold cross evaluation to figure out one optimal parameter based on default value; In other word, set one parameter as a variable and the rest of parameter as constants. The step is to narrow the range of input parameters.
3. repeat #2 k times to get approximate optimal parameters of range (k is the number of parameters to fine turning)
4. combine every improved parameter together then evaluate the model and output the result
5. random input parameters in the decision range to see whether get a better performance

(please see the attached file for more details)

1. Hyperparameters & Error rates