Xgboost

What is Xgboost?

Extreme Gradient Boosting (xgboost) is similar to gradient boosting framework but more efficient. It has both linear model solver and tree learning algorithms. So, what makes it fast is its capacity to do parallel computation on a single machine.

This makes xgboost at least 10 times faster than existing gradient boosting implementations. It supports various objective functions, including regression, classification and ranking.

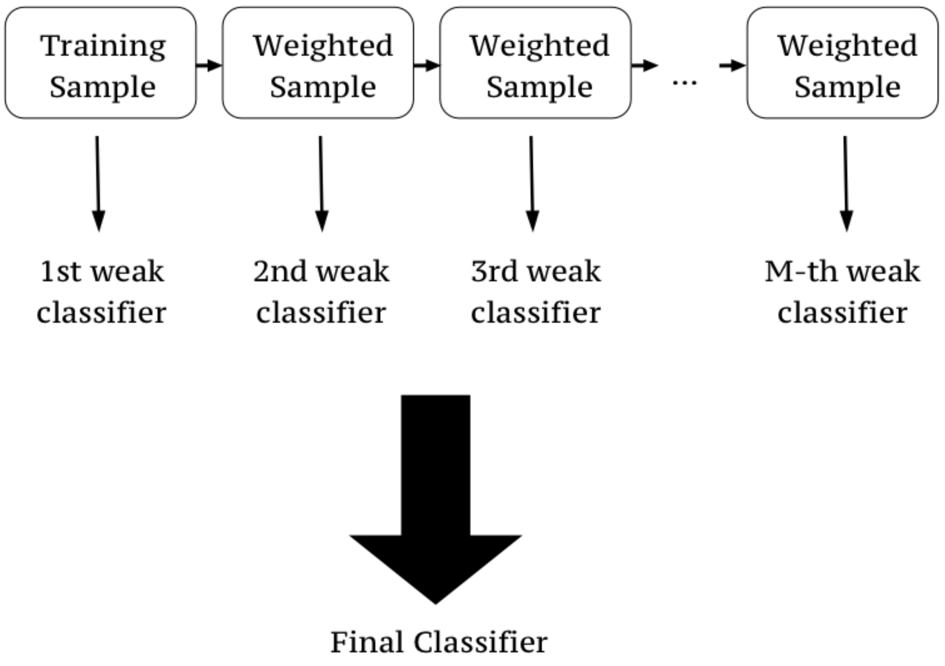
Since it is very high in predictive power but relatively slow with implementation, “xgboost” becomes an ideal fit for many competitions. It also has additional features for doing cross validation and finding important variables.

Idea of boosting

Let's start with intuitive definition of the concept:

> **Boosting** (*Freud and Shapire, 1996*) - algorithm allowing to fit **many** weak classifiers to **reweighted** versions of the training data. Classify final examples by majority voting.

When using boosting techinque all instance in dataset are assigned a score that tells *how difficult to classify* they are. In each following iteration the algorithm pays more attention (assign bigger weights) to instances that were wrongly classified previously.



In the first iteration all instance weights are equal.

Ensemble parameters are optimized in **stagewise way** which means that we are calculating optimal parameters for the next classifier holding fixed what was already calculated. This might sound like a limitation but turns out it's a very resonable way of regularizing the model.

Weak classifier - why tree?

First what is a weak classifier?

> **Weak classifier** - an algorithm **slightly better** than random guessing.

Every algorithm can be used as a base for boosting techinique, but trees have some nice properties that makes them more suitable candidates.

Pro's

- computational scalability,
- handling missing values,
- robust to outliers,
- does not require feature scalling,
- can deal with irrelevant inputs,
- interpretable (if small),
- can handle mixed predictors (quantitive and qualitative)

Con's

- can't extract linear combination of features
- small predictive power (high variance)

Boosting techinque can try to reduce the variance by **averaging** many **different** trees (where each one is solving the same problem)

Common Algorithms

In every machine learning model the training objective is a sum of a loss function L and regularization Ω:

$$obj = L + \Omega$$

The loss function controls the predictive power of an algorithm and regularization term controls it's simplicity.

AdaBoost (Adaptive Boosting)

The implementation of boosting technique using decision tress (it's a *meta-estimator* which means you can fit any classifier in). The intuitive recipie is presented below:

Algorithm:

Assume that the number of training samples is denoted by N, and the number of iterations (created trees) is M. Notice that possible class outputs are $Y = \{-1, 1\}$

1. Initialize the observation weights $w_i = \frac{1}{N}$ where $i = 1, 2, \ldots, N$
2. For $m = 1$ to M:
    - fit a classifier $G_m(x)$ to the training data using weights w_i,
    - compute $err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x))}{\sum_{i=1}^{N} w_i}$,
    - compute $\alpha_m = \log((1 - err_m)/err_m)$,
    - set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x)]$, where $i = 1, 2, \ldots, N$
3. Output $G_m(x) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$

Generalized Boosted Models

We can take advantage of the fact that the loss function can be represented with a form suitable for optimalization (due to the stage-wise additivity). This creates a class of general boosting algorithms named simply **generalized boosted model (GBM)**.

An example of a GBM is **Gradient Boosted Tree** which uses decision tree as an estimator. It can work with different loss functions (regression, classification, risk modeling etc.), evaluate it's gradient and approximates it with a simple tree (stage-wisely, that minimizes the overall error).

AdaBoost is a special case of Gradient Boosted Tree that uses exponential loss function.

How XGBoost helps

The problem with most tree packages is that they don't take regularization issues very seriously - they allow to grow many very similar trees that can be also sometimes quite bushy.

GBT tries to approach this problem by adding some regularization parameters. We can:

- control tree structure (maximum depth, minimum samples per leaf),
- control learning rate (shrinkage),
- reduce variance by introducing randomness (stochastic gradient boosting - using random subsamples of instances and features)

But it could be improved even further. Enter XGBoost.

> **XGBoost** (*extreme gradient boosting*) is a **more regularized** version of Gradient Boosted Trees.

It was develop by Tianqi Chen in C++ but also enables interfaces for Python, R, Julia.

The main advantages:

- good bias-variance (simple-predictive) trade-off "out of the box",
- great computation speed,
- package is evolving (author is willing to accept many PR from community)

XGBoost's objective function is a sum of a specific loss function evaluated over all predictions and a sum of regularization term for all predictors (K trees). In the formula f_k means a prediction coming from k-th tree.

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

Loss function depends on the task being performed (classification, regression, etc.) and a regularization term is described by the following equation:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

First part (γT) is responsible for controlling the overall number of created leaves, and the second term ($\frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2$) watches over the their's scores.

Implementation

In [1]:

```
import numpy as np
import xgboost as xgb
```

```
/usr/local/lib/python3.5/dist-packages/sklearn/cross_validation.py:44: Dep
recationWarning: This module was deprecated in version 0.18 in favor of th
e model_selection module into which all the refactored classes and functio
ns are moved. Also note that the interface of the new CV iterators are dif
ferent from that of this module. This module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

Loading data

We are going to use bundled Agaricus (https://archive.ics.uci.edu/ml/datasets/Mushroom) dataset which can be downloaded here (https://github.com/dmlc/xgboost/tree/master/demo/data).

> This data set records biological attributes of different mushroom species, and the target is to predict whether it is poisonous
>
> This data set includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom;

It consist of 8124 instances, characterized by 22 attributes (both numeric and categorical). The target class is either 0 or 1 which means binary classification problem.

> **Important**: XGBoost handles only numeric variables.

Lucily all the data have alreay been pre-process for us. Categorical variables have been encoded, and all instances divided into train and test datasets. You will know how to do this on your own in later lectures.

Data needs to be stored in `DMatrix` object which is designed to handle sparse datasets. It can be populated in couple ways:

- using libsvm format txt file,
- using Numpy 2D array (most popular),
- using XGBoost binary buffer file

In this case we'll use first option.

> Libsvm files stores only non-zero elements in format
>
> `<label> <feature_a>:<value_a> <feature_c>:<value_c> ... <feature_z>:<value_z>`
>
> Any missing features indicate that it's corresponding value is 0.

In [2]:

```
dtrain = xgb.DMatrix('data/agaricus.txt.train')
dtest = xgb.DMatrix('data/agaricus.txt.test')
```

Let's examine what was loaded:

In [3]:

```
print("Train dataset contains {0} rows and {1} columns".format(dtrain.num_row(), dtrain
.num_col()))
print("Test dataset contains {0} rows and {1} columns".format(dtest.num_row(), dtest.nu
m_col()))
```

```
Train dataset contains 6513 rows and 127 columns
Test dataset contains 1611 rows and 127 columns
```

In [4]:

```
print("Train possible labels: ")
print(np.unique(dtrain.get_label()))

print("\nTest possible labels: ")
print(np.unique(dtest.get_label()))
```

```
Train possible labels: 
[ 0.  1.]

Test possible labels: 
[ 0.  1.]
```

Specify training parameters

Let's make the following assuptions and adjust algorithm parameters to it:

- we are dealing with binary classification problem (`'objective':'binary:logistic'`),
- we want shallow single trees with no more than 2 levels (`'max_depth':2`),
- we don't any oupout (`'silent':1`),
- we want algorithm to learn fast and aggressively (`'eta':1`),
- we want to iterate only 5 rounds

In [5]:

```
params = {
    'objective':'binary:logistic',
    'max_depth':2,
    'silent':1,
    'eta':1
}

num_rounds = 5
```

Training classifier

To train the classifier we simply pass to it a training dataset, parameters list and information about number of iterations.

In [6]:

```
bst = xgb.train(params, dtrain, num_rounds)
```

We can also observe performance on test dataset using `watchlist`

In [7]:

```
watchlist  = [(dtest,'test'), (dtrain,'train')] # native interface only
bst = xgb.train(params, dtrain, num_rounds, watchlist)
```

```
[0]     test-error:0.042831     train-error:0.046522
[1]     test-error:0.021726     train-error:0.022263
[2]     test-error:0.006207     train-error:0.007063
[3]     test-error:0.018001     train-error:0.0152
[4]     test-error:0.006207     train-error:0.007063
```

Make predictions

In [8]:

```
preds_prob = bst.predict(dtest)
preds_prob
```

Out[8]:

```
array([ 0.08073306,  0.92217326,  0.08073306, ...,  0.98059034,
        0.01182149,  0.98059034], dtype=float32)
```

Calculate simple accuracy metric to verify the results. Of course validation should be performed accordingly to the dataset, but in this case accuracy is sufficient.

In [9]:

```
labels = dtest.get_label()
preds = preds_prob > 0.5 # threshold
correct = 0

for i in range(len(preds)):
    if (labels[i] == preds[i]):
        correct += 1

print('Predicted correctly: {0}/{1}'.format(correct, len(preds)))
print('Error: {0:.4f}'.format(1-correct/len(preds)))
```

```
Predicted correctly: 1601/1611
Error: 0.0062
```

Using Scikit-learn Interface

The following notebook presents the alternative approach for using XGBoost algorithm.

Loading libraries

Begin with loading all required libraries.

In [10]:

```
import numpy as np

from sklearn.datasets import load_svmlight_files
from sklearn.metrics import accuracy_score

from xgboost.sklearn import XGBClassifier
```

Loading data

We are going to use the same dataset as in previous lecture. The scikit-learn package provides a convenient function `load_svmlight` capable of reading many libsvm files at once and storing them as Scipy's sparse matrices.

In [12]:

```
X_train, y_train, X_test, y_test = load_svmlight_files(('data/agaricus.txt.train', 'dat
a/agaricus.txt.test'))
```

Examine what was loaded

Specify training parameters

All the parameters are set like in the previous example

- we are dealing with binary classification problem (`'objective':'binary:logistic'`),
- we want shallow single trees with no more than 2 levels (`'max_depth':2`),
- we don't any oupout (`'silent':1`),
- we want algorithm to learn fast and aggressively (`'learning_rate':1`), (in naive named `eta`)
- we want to iterate only 5 rounds (`n_estimators`)

In [15]:

```
params = {
    'objective': 'binary:logistic',
    'max_depth': 2,
    'learning_rate': 1.0,
    'silent': 1.0,
    'n_estimators': 5
}
```

Training classifier

In [16]:

```
bst = XGBClassifier(**params).fit(X_train, y_train)
```

Make predictions

In [17]:

```
preds = bst.predict(X_test)
preds
```

Out[17]:

```
array([ 0.,  1.,  0., ...,  1.,  0.,  1.])
```

Calculate obtained error

In [18]:

```
correct = 0

for i in range(len(preds)):
    if (y_test[i] == preds[i]):
        correct += 1

acc = accuracy_score(y_test, preds)

print('Predicted correctly: {0}/{1}'.format(correct, len(preds)))
print('Error: {0:.4f}'.format(1-acc))
```

```
Predicted correctly: 1601/1611
Error: 0.0062
```

Evaluate results

Specify training parameters - we are going to use 5 decision tree stumps with average learning rate.

In [21]:

```
# specify general training parameters
params = {
    'objective':'binary:logistic',
    'max_depth':1,
    'silent':1,
    'eta':0.5
}

num_rounds = 5
```

Before training the model let's also specify `watchlist` array to observe it's performance on the both datasets.

In [22]:

```
watchlist  = [(dtest,'test'), (dtrain,'train')]
```

Using predefined evaluation metrics

What is already available?

There are already some predefined metrics availabe. You can use them as the input for the `eval_metric` parameter while training the model.

- `rmse` - root mean square error (https://www.wikiwand.com/en/Root-mean-square_deviation),
- `mae` - mean absolute error (https://en.wikipedia.org/wiki/Mean_absolute_error?oldformat=true),
- `logloss` - negative log-likelihood (https://en.wikipedia.org/wiki/Likelihood_function?oldformat=true)
- `error` - binary classification error rate. It is calculated as `#(wrong cases)/#(all cases)`. Treat predicted values with probability $p > 0.5$ as positive,
- `merror` - multiclass classification error rate. It is calculated as `#(wrong cases)/#(all cases)`,
- `auc` - area under curve (https://en.wikipedia.org/wiki/Receiver_operating_characteristic?oldformat=true),
- `ndcg` - normalized discounted cumulative gain (https://en.wikipedia.org/wiki/Discounted_cumulative_gain?oldformat=true),
- `map` - mean average precision (https://en.wikipedia.org/wiki/Information_retrieval?oldformat=true)

By default an `error` metric will be used.

In [23]:

```
bst = xgb.train(params, dtrain, num_rounds, watchlist)
```

```
[0]     test-error:0.11049      train-error:0.113926
[1]     test-error:0.11049      train-error:0.113926
[2]     test-error:0.03352      train-error:0.030401
[3]     test-error:0.027312     train-error:0.021495
[4]     test-error:0.031037     train-error:0.025487
```

To change is simply specify the `eval_metric` argument to the `params` dictionary.

In [24]:

```
params['eval_metric'] = 'logloss'
bst = xgb.train(params, dtrain, num_rounds, watchlist)
```

```
[0]     test-logloss:0.457887   train-logloss:0.460108
[1]     test-logloss:0.383911   train-logloss:0.378728
[2]     test-logloss:0.312678   train-logloss:0.308061
[3]     test-logloss:0.26912    train-logloss:0.26139
[4]     test-logloss:0.239746   train-logloss:0.232174
```

You can also use multiple evaluation metrics at one time

In [25]:

```
params['eval_metric'] = ['logloss', 'auc']
bst = xgb.train(params, dtrain, num_rounds, watchlist)
```

```
[0]	test-logloss:0.457887	test-auc:0.892138	train-logloss:0.46
0108	train-auc:0.888997
[1]	test-logloss:0.383911	test-auc:0.938901	train-logloss:0.37
8728	train-auc:0.942881
[2]	test-logloss:0.312678	test-auc:0.976157	train-logloss:0.30
8061	train-auc:0.981415
[3]	test-logloss:0.26912	test-auc:0.979685	train-logloss:0.26
139	train-auc:0.985158
[4]	test-logloss:0.239746	test-auc:0.9785 train-logloss:0.232174	tr
ain-auc:0.983744
```

Creating custom evaluation metric

In order to create our own evaluation metric, the only thing needed to do is to create a method taking two arguments - predicted probabilities and `DMatrix` object holding training data.

In this example our classification metric will simply count the number of misclassified examples assuming that classes with $p > 0.5$ are positive. You can change this threshold if you want more certainty.

The algorithm is getting better when the number of misclassified examples is getting lower. Remember to also set the argument `maximize=False` while training.

In [26]:

```
# custom evaluation metric
def misclassified(pred_probs, dtrain):
    labels = dtrain.get_label() # obtain true labels
    preds = pred_probs > 0.5 # obtain predicted values
    return 'misclassified', np.sum(labels != preds)
```

In [27]:

```
bst = xgb.train(params, dtrain, num_rounds, watchlist, feval=misclassified, maximize=Fa
lse)
```

```
[0]	test-misclassified:178	train-misclassified:742
[1]	test-misclassified:178	train-misclassified:742
[2]	test-misclassified:54	train-misclassified:198
[3]	test-misclassified:44	train-misclassified:140
[4]	test-misclassified:50	train-misclassified:166
```

Handle Imbalanced Dataset

There are plenty of examples in real-world problems that deals with imbalanced target classes. Imagine medical data where there are only a few positive instances out of thousands of negatie (normal) ones. Another example might be analyzing fraud transaction, in which the actual frauds represent only a fraction of all available data.

General advices

These are some common tactics when approaching imbalanced datasets:

- collect more data,
- use better evaluation metric (that notices mistakes - ie. AUC, F1, Kappa, ...),
- try oversampling minority class or undersampling majority class,
- generate artificial samples of minority class (ie. SMOTE algorithm)

In XGBoost you can try to:

- make sure that parameter `min_child_weight` is small (because leaf nodes can have smaller size groups), it is set to `min_child_weight=1` by default,
- assign more weights to specific samples while initalizing `DMatrix`,
- control the balance of positive and negative weights using `set_pos_weight` parameter,
- use AUC for evaluation

```
In [ ]:
```