**Boost Your ML skills with XGBoost**

Introduction :

In this blog we will discuss one of the Popular Boosting Ensemble algorithm called XGBoost.

XGBoost is the most popular machine learning algorithm these days. Regardless of the data type (regression or classification), it is well known to provide better solutions than other ML algorithms.

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.

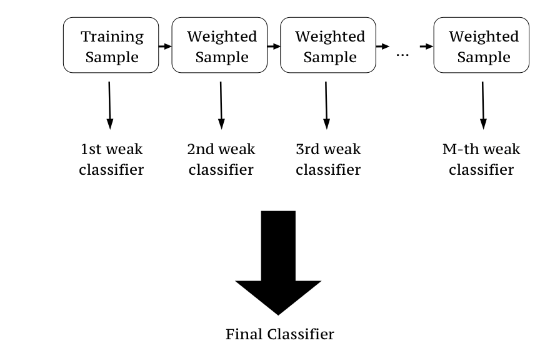
Since it is very high in predictive power but relatively slow with implementation, “xgboost” becomes an ideal fit for many competitions. It also has additional features for doing cross validation and finding important variables.

### **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.

#### **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)

### **How XGBoost helps [¶](http://localhost:8889/notebooks/Day5-XGboost-KNN-NB/Xgboost.ipynb#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.

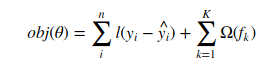
It was develop by Tianqi Chen in C++ but also enables interfaces for Python, R, Julia.

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 (KK trees).

Mathematically, it can be represented as :



XGBoost handles only numeric variables.

**Problem Statement :**

To build a simple boosting classification model called XGBoost , for predicting the quality of the car given few of other car attributes.

**Data details**

|  |
| --- |
| ========================================== 1. Title: Car Evaluation Database  ==========================================  The dataset is available at “<http://archive.ics.uci.edu/ml/datasets/Car+Evaluation>”  2. Sources:  (a) Creator: Marko Bohanec  (b) Donors: Marko Bohanec (marko.bohanec@ijs.si)  Blaz Zupan (blaz.zupan@ijs.si)  (c) Date: June, 1997  3. Past Usage:   The hierarchical decision model, from which this dataset is  derived, was first presented in    M. Bohanec and V. Rajkovic: Knowledge acquisition and explanation for  multi-attribute decision making. In 8th Intl Workshop on Expert  Systems and their Applications, Avignon, France. pages 59-78, 1988.   Within machine-learning, this dataset was used for the evaluation  of HINT (Hierarchy INduction Tool), which was proved to be able to  completely reconstruct the original hierarchical model. This,  together with a comparison with C4.5, is presented in   B. Zupan, M. Bohanec, I. Bratko, J. Demsar: Machine learning by  function decomposition. ICML-97, Nashville, TN. 1997 (to appear)  4. Relevant Information Paragraph:   Car Evaluation Database was derived from a simple hierarchical  decision model originally developed for the demonstration of DEX  (M. Bohanec, V. Rajkovic: Expert system for decision  making. Sistemica 1(1), pp. 145-157, 1990.). The model evaluates  cars according to the following concept structure:   CAR car acceptability  . PRICE overall price  . . buying buying price  . . maint price of the maintenance  . TECH technical characteristics  . . COMFORT comfort  . . . doors number of doors  . . . persons capacity in terms of persons to carry  . . . lug\_boot the size of luggage boot  . . safety estimated safety of the car   Input attributes are printed in lowercase. Besides the target  concept (CAR), the model includes three intermediate concepts:  PRICE, TECH, COMFORT. Every concept is in the original model  related to its lower level descendants by a set of examples (for  these examples sets see<http://www-ai.ijs.si/BlazZupan/car.html).>   The Car Evaluation Database contains examples with the structural  information removed, i.e., directly relates CAR to the six input  attributes: buying, maint, doors, persons, lug\_boot, safety.   Because of known underlying concept structure, this database may be  particularly useful for testing constructive induction and  structure discovery methods.  5. Number of Instances: 1728  (instances completely cover the attribute space)  6. Number of Attributes: 6  7. Attribute Values:   buying v-high, high, med, low  maint v-high, high, med, low  doors 2, 3, 4, 5-more  persons 2, 4, more  lug\_boot small, med, big  safety low, med, high  8. Missing Attribute Values: none  9. Class Distribution (number of instances per class)   class N N[%]  -----------------------------  unacc 1210 (70.023 %)   acc 384 (22.222 %)   good 69 ( 3.993 %)   v-good 65 ( 3.762 %) |

Tools to be used :

Numpy,pandas,scikit-learn

**Python Implementation with code :**

**Import necessary libraries**

Import the necessary modules from specific libraries.

|  |
| --- |
| import os  import numpy as np, pandas as pd  import matplotlib.pyplot as plt  from sklearn import metrics, model\_selection  from xgboost.sklearn import XGBClassifier |

**Load the data set**

Use pandas module to read the bike data from the file system. Check few records of the dataset.

|  |
| --- |
| data = pd.read\_csv('data/car\_quality/car.data',names=['buying','maint','doors','persons','lug\_boot','safety','class'])  data.head()  buying maint doors persons lug\_boot safety class  0 vhigh vhigh 2 2 small low unacc  1 vhigh vhigh 2 2 small med unacc  2 vhigh vhigh 2 2 small high unacc  3 vhigh vhigh 2 2 med low unacc  4 vhigh vhigh 2 2 med med unacc |

**Check few information about the data set**

|  |
| --- |
| data.info()  <class 'pandas.core.frame.DataFrame'> RangeIndex: 1728 entries, 0 to 1727 Data columns (total 7 columns): buying 1728 non-null object maint 1728 non-null object doors 1728 non-null object persons 1728 non-null object lug\_boot 1728 non-null object safety 1728 non-null object class 1728 non-null object dtypes: object(7) memory usage: 94.6+ KB |

The train data set has 1728 rows and 7 columns.

There are no missing values in the dataset

**Identify the target variable**

|  |
| --- |
| data['class'],class\_names = pd.factorize(data['class']) |

The target variable is marked as class in the dataframe. The values are present in string format. However the algorithm requires the variables to be coded into its equivalent integer codes. We can convert the string categorical values into a integer code using factorize method of the pandas library.

Let’s check the encoded values now.

|  |
| --- |
| print(class\_names)  print(data['class'].unique())  Index([u'unacc', u'acc', u'vgood', u'good'], dtype='object') [0 1 2 3] |

As we can see the values has been encoded into 4 different numeric labels.

**Identify the predictor variables and encode any string variables to equivalent integer codes**

|  |
| --- |
| data['buying'],\_ = pd.factorize(data['buying'])  data['maint'],\_ = pd.factorize(data['maint'])  data['doors'],\_ = pd.factorize(data['doors'])  data['persons'],\_ = pd.factorize(data['persons'])  data['lug\_boot'],\_ = pd.factorize(data['lug\_boot'])  data['safety'],\_ = pd.factorize(data['safety'])  data.head()  buying maint doors persons lug\_boot safety class  0 0 0 0 0 0 0 0  1 0 0 0 0 0 1 0  2 0 0 0 0 0 2 0  3 0 0 0 0 1 0 0  4 0 0 0 0 1 1 0 |

Check the data types now :

|  |
| --- |
| data.info()  <class 'pandas.core.frame.DataFrame'> RangeIndex: 1728 entries, 0 to 1727 Data columns (total 7 columns): buying 1728 non-null int64 maint 1728 non-null int64 doors 1728 non-null int64 persons 1728 non-null int64 lug\_boot 1728 non-null int64 safety 1728 non-null int64 class 1728 non-null int64 dtypes: int64(7) memory usage: 94.6 KB |

Everything is now converted in integer form.

**Select the predictor feature and select the target variable**

|  |
| --- |
| X = data.iloc[:,:-1]  y = data.iloc[:,-1] |

**Train test split :**

|  |
| --- |
| # split data randomly into 70% training and 30% test  X\_train, X\_test, y\_train, y\_test = model\_selection.train\_test\_split(X, y, test\_size=0.3, random\_state=123) |

**Training / model fitting**

|  |
| --- |
| params = {  'objective': 'binary:logistic',  'max\_depth': 2,  'learning\_rate': 1.0,  'silent': 1.0,  'n\_estimators': 5  }  model = XGBClassifier(\*\*params).fit(X\_train, y\_train) |

**Model parameters study :**

|  |
| --- |
| # use the model to make predictions with the test data  y\_pred = model.predict(X\_test)  # how did our model perform?  count\_misclassified = (y\_test != y\_pred).sum()  print('Misclassified samples: {}'.format(count\_misclassified))  accuracy = metrics.accuracy\_score(y\_test, y\_pred)  print('Accuracy: {:.2f}'.format(accuracy))  Misclassified samples: 58 Accuracy: 0.89 |

The model actually has a 89% accuracy score,Not bad at all. There you have it. That’s how to implement your first xgboost model with scikit-learn. Load your favorite data set and give it a try!

Algo Advantages :

**Parallel Computing:** It is enabled with parallel processing (using OpenMP); i.e., when you run xgboost, by default, it would use all the cores of your laptop/machine.

**Regularization**: I believe this is the biggest advantage of xgboost. GBM has no provision for regularization. Regularization is a technique used to avoid overfitting in linear and tree-based models.

**Enabled Cross Validation**: In R, we usually use external packages such as caret and mlr to obtain CV results. But, xgboost is enabled with internal CV function (we'll see below).

**Missing Values**: XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that if there exists any trend in missing values, it is captured by the model.

**Flexibility**: In addition to regression, classification, and ranking problems, it supports user-defined objective functions also. An objective function is used to measure the performance of the model given a certain set of parameters. Furthermore, it supports user defined evaluation metrics as well.

**Availability**: Currently, it is available for programming languages such as R, Python, Java, Julia, and Scala.

**Save and Reload:** XGBoost gives us a feature to save our data matrix and model and reload it later. Suppose, we have a large data set, we can simply save the model and use it in future instead of wasting time redoing the computation.

**Tree Pruning**: Unlike GBM, where tree pruning stops once a negative loss is encountered, XGBoost grows the tree upto max\_depth and then prune backward until the improvement in loss function is below a threshold.