

# ***Ensembles of classifiers: Bagging and Boosting***

***Edgar Acuña***

***Department of Mathematical Sciences  
University of Puerto Rico –Mayaguez  
[academic.uprm.edu/eacuna](http://academic.uprm.edu/eacuna)***

***Noviembre 2021***

# The Misclassification Error

Let  $C(x, L)$  be the classifier constructed by using the training sample  $L$ ., and  $T$  another large sample from the same population as  $L$  was drawn from, then the misclassification error (ME) of the classifier  $C$  is the proportion of misclassified cases of  $T$  using  $C$ .

The ME can be descomposed as

$$ME(C) = ME(C^*) + Bias^2(C) + Var(C)$$

where  $C^*(x) = \operatorname{argmax}_j P(Y=j/X=x)$  (Bayes Classifier)

Methods to estimate ME: Resubstitution, Crossvalidation, Bootstrapping

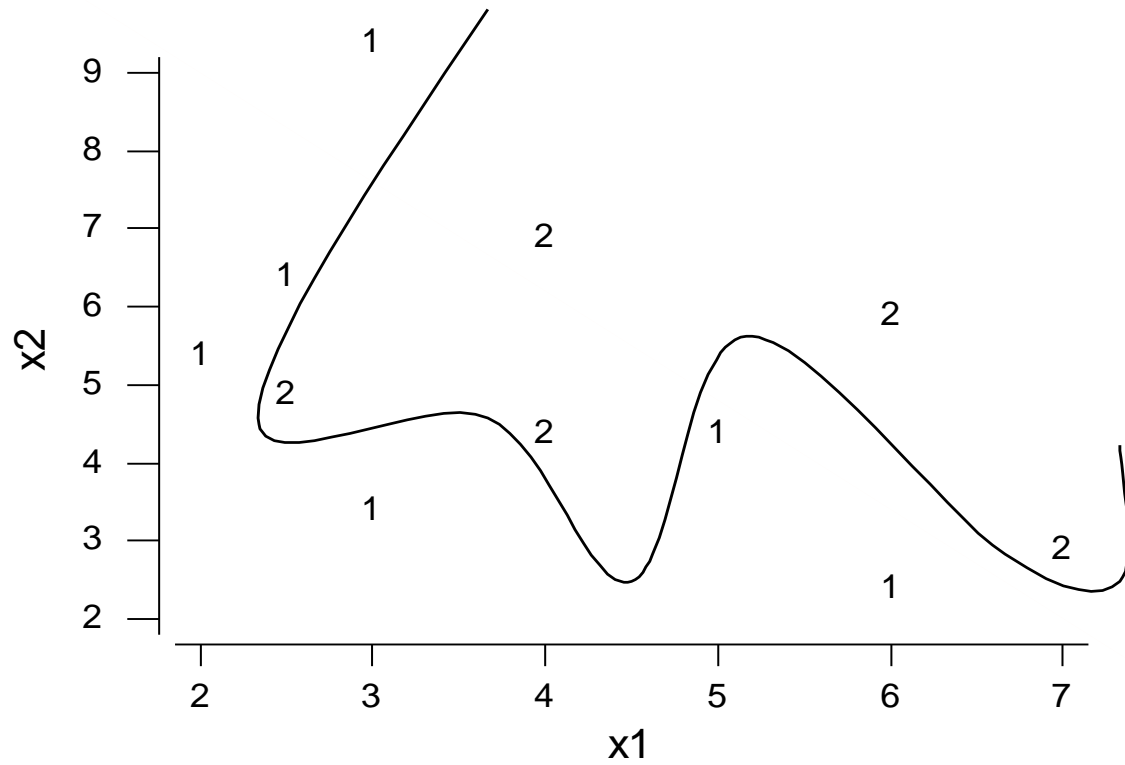
The classifier may either overfit the data (Low bias and large variance) or underfit the data ( Large bias and small variance).

Breiman (1996) heuristically defines a classifier as unstable if a small change in the data  $L$  can make large changes in the classification. Unstable classifiers have low bias but high variance.

CART and Neural networks are unstable classifiers.

Linear discriminant analysis and K-nearest neighbor classifiers are stable.

# Overfitting



# Combination of classifiers (Ensembles)

It is possible to reduce the bias and variance of the misclassification error by combining the prediction of several classifiers. This combination is called an **Ensemble** and in general is more precise than the individual classifiers. Among the methods used to build ensembles are the following:

**Bagging** (Bootstrap aggregating by Breiman, 1996)

**AdaBoosting** (Adaptive Boosting by Freund and Schapire, 1996)

**Arcing** (Adaptively resampling and combining, by Breiman (1998).

**Gradient Boosting** (Friedman, 1999)

**XGboosting** (Chen, 2014)

# Bootstrap Sample

A bootstrap sample of a training sample  $L$  is a sample with replacement taken from  $L$  and of the same size than  $L$

```
x=[5,3,12,13,21,31,8,9,15,17,24,32] # This is the original sample
```

```
boot1=np.random.choice(x,12) #Extracting one bootstrap sample
```

```
print boot1
```

```
[31 8 5 31 24 13 21 3 32 9 9 17]
```

```
np.unique(boot1)
```

```
array([ 3, 5, 8, 9, 13, 17, 21, 24, 31, 32])
```

```
boot2=np.random.choice(x,12) #Second bootstrap sample
```

```
print boot2
```

```
[ 5 12 12 13 13 15 9 13 24 15 21 12]
```

```
np.unique(boot2)
```

```
array([ 5, 9, 12, 13, 15, 21, 24])
```

Note: Approximately, 37% of the instances of the training sample  $L$  DO NOT appear in any bootstrap sample. In the above examples 16.67% and 41.67% of instances do not appear in each of the bootstrap samples.

# The Bagging Algorithm: Breiman (1996)

**Input:** learning set  $\mathcal{L}$ , classifier C, integer T (number of bootstrap samples), J classes.

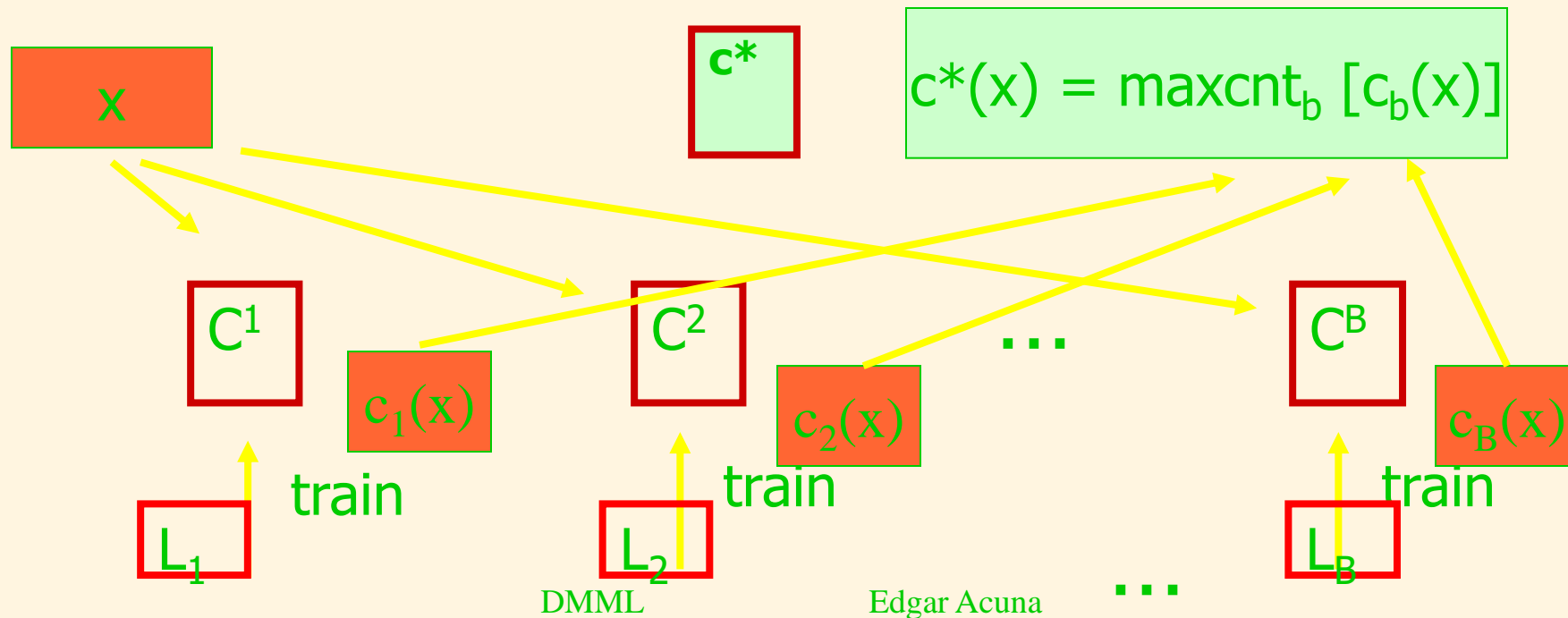
1. For  $i=1$  to T {
2.  $B_i =$  Bootstrap sample from  $\mathcal{L}$  (sample with replacement)
3.  $C_i = C(B_i)$
4. }
5.  $C_A(x) = \underset{j \in \{1, \dots, J\}}{\operatorname{argmax}} \sum_{i: C_i(x)=j} 1$  (The class most voted)

**Output:** Ensemble  $C_A$



# Bagging with cross validation

- From the training sample  $L$  select  $B$  random samples with replacement (bootstrap samples) obtaining  $B$  different training samples  $L_1, \dots, L_B$  of size  $N$ .
- For each sample  $L_b$  a classifier  $C^b$  is built.
- Using 10-fold cross validation, each case  $x$  of  $L$  is assigned to the class  $c^*(x)=j$  by voting.





# Bagging using scikit-learn

```
url= "http://academic.uprm.edu/eacuna/diabetes.dat"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = pd.read_table(url, names=names, header=None)
#The response variable must be binary (0,1)
y=data['class']-1
X=data.iloc[:,0:8]
modeltree = tree.DecisionTreeClassifier()
bagging = BaggingClassifier(modeltree,n_estimators=100)
# Accuracy rate
bagging.fit(X, y)
predictions = bagging.predict(X)
print(classification_report(y, predictions))
#Estimating the accuracy by cross validation
kfold = model_selection.KFold(n_splits=10, random_state=99)
results = model_selection.cross_val_score(bagging, X, y, cv=kfold)
print(results.mean())
0.756459330144
```

# Out-of-Bag (OOB) error estimation

Assuming that  $K=100$  bootstrap sample are drawn from the original sample  $L$ , then an instance of  $L$  will not be chosen in approximately 37 percent of these samples, therefore they can be considered as a test sample  $T$ .

Hence any instance would have something like 37 predictions. Using voting we can have a final prediction.

The missclassification error rate estimated Out-of-bag is the average of the number of errors using those final predictions.

## Out-of-bag error estimation using scikit-learn

```
bagging1 = BaggingClassifier(modeltree,n_estimators=100, oob_score=True)
bagging1.fit(X, y)
bagging1.oob_score_  
0.7643229166666666
```

# Boosting

In this method, also bootstrap samples are chosen but given a selection weight to each instance. In the first bootstrap sample, all the instances have the same weight  $1/N$  to be chosen, where  $N$  is the number of instances.

In the following steps this weight changes in such way the instances classified incorrectly in the previous sample have more weight to be selected in the next step. The instances correctly classified have less weight to be chosen in a later step.

The goal is to force the classifier to learn to classify correctly an instance, which is not easy to classify.

There are plenty versions of boosting, the first one is known as AdaBoosting. But the ones that are being used at the present are Gradient Boosting and XGBoost

# Adaboosting Algorithm: Freund & Schapire[1996]

**Input:** Learning set  $L$  , classifier  $C$ , integers  $N, T$

1.  $B = L$  with weights  $w_1(x_j) = 1/N$ . For  $j = 1, \dots, N$

2. For  $i = 1$  to  $T$  {  $C_i = C(B)$

3. Set  $e_i = \sum_{x_j \in B: C_i(x_j) \neq y_j} w_i(x_j)$  . If either  $e_i > 1/2$  or  $e_i = 0$  then restart assigning equal weights.

4. Set  $\beta_i = e_i / (1 - e_i)$

5. Update the weights: for each  $x_j \in B$ , if  $C_i(x_j) \neq y_j$  then  $w_{i+1}(x_j) = w_i(x_j) / 2e_i$  , else  $w_{i+1}(x_j) = w_i(x_j) / 2(1 - e_i)$  }

6.  $C^*(x) = \arg \max_{j \in \{1, \dots, J\}} \sum_{i: C_i(x) = j} \log \frac{1}{\beta_i}$

**Output:** Ensemble  $C^*$

# Boosting

Procedure:

1. Initially, all observations are given equal weights.
2. A classifier is built on a subset of data.
3. Using this classifier, predictions are made on the whole dataset.
4. Errors are calculated by comparing the predictions and actual values.
5. While creating the next classifier, higher weights are given to the data points which were predicted incorrectly.
6. Weights can be determined using the error value. For instance, the higher the error the more is the weight assigned to the observation.
7. This process is repeated until the error function does not change, or the maximum limit of the number of estimators is reached.

## AdaBoosting using scikit-learn

```
adaboost = AdaBoostClassifier(modeltree,n_estimators=100,learning_rate=1)
```

```
adaboost.fit(X, y)
```

```
predictions = adaboost.predict(X)
```

```
print(classification_report(y, predictions))
```

```
      precision recall f1-score support
0      1.00      1.00      1.00      500
1      1.00      1.00      1.00      268
avg / total 1.00 1.00 1.00      768
```

```
kfold = model_selection.KFold(n_splits=10, random_state=999)
```

```
results = model_selection.cross_val_score(adaboost, X, y, cv=kfold)
```

```
print(results.mean())
```

```
0.7252734108
```



# Gradient Boosting

Procedure:

1. A classifier is built on a subset of data.
2. Using this classifier, predictions are made on the whole dataset.
3. Errors are calculated by comparing the predictions and actual values.
4. A new classifier is created using the errors calculated in step 3 as target variable.
5. The predictions made by this new classifier are combined with the predictions of the previous. This is justified only if a quadratic loss function is used, since the derivative of a quadratic function is a linear function.
6. New errors are calculated using this predicted value and actual value.
7. This process is repeated until the error loss function does not change, or the maximum limit of the number of estimators is reached.

## Gradient Boosting using scikit-learn

```
gboost = GradientBoostingClassifier(n_estimators=100)
gboost.fit(X, y)
predictions = gboost.predict(X)
print(classification_report(y, predictions))
```

	precision	recall	f1-score	support
0	0.90	0.96	0.93	500
1	0.91	0.81	0.86	268
avg / total	0.91	0.91	0.90	768

```
kfold = model_selection.KFold(n_splits=10, random_state=999)
results = model_selection.cross_val_score(gboost, X, y, cv=kfold)
print(results.mean())
```

0.766900205058

# Gradient Boosting using h2o

```
diabetes = h2o.import_file("https://academic.uprm.edu/eacuna/diabetes.dat")
myx=['C1','C2','C3','C4','C5','C6','C7','C8']
diabetes['C9']=diabetes['C9'].asfactor()
myy="C9"
gbm1 = H2OGradientBoostingEstimator(model_id="gbm_covType_v1",ntrees = 100,
max_depth=3,nfolds=10, sample_rate = 1,col_sample_rate = 1,seed=2000000
)
gbm1.train(myx, myy, training_frame=diabetes)
y_pred=gbm1.predict(diabetes)
print (y_pred['predict']==diabetes['C9']).mean()
[0.9010416666666666]
```

# Previous results on combining classifiers

Reference	Classifier	Relative Improv. (%)	
		Bagging	Boosting
Breiman (1996)	<b>CART</b>	<b>29.0</b>	<b>-----</b>
Freund & Schapire (1996)	<b>C4.5</b>	<b>20.0</b>	<b>24.8</b>
Quinlan (1996)	<b>C4.5</b>	<b>10.0</b>	<b>15.01</b>
Maclin & Opitz (1997)	<b>C4.5</b>	<b>18.5</b>	<b>22.0</b>
Maclin & Opitz (1997)	<b>Neural Net</b>	<b>13.3</b>	<b>17.1</b>
Breiman (1998)	<b>CART</b>	<b>36.0</b>	<b>48.4</b>
Bauer & Kohavi (1999)	<b>MC4</b>	<b>14.5</b>	<b>27.0</b>
Dietterich(2000)	<b>C4.5</b>	<b>16.9</b>	<b>22.4</b>
Daza (2002)	<b>G.M.</b>	<b>10.1</b>	<b>0.8</b>
Acuna & Rojas (2002)	<b>Kernel</b>	<b>4.9</b>	<b>1.9</b>

DMML

Edgar Acuna

- Bagging reduces variance. AdaBoosting reduces both bias and variance
- Bagging can be parallelized easily, but Boosting is essentially sequential and only some part of the algorithm can be parallelized.
- Boosting is only useful for large sample datasets and for classifiers that perform poorly.

# XGBoosting (Extreme Gradient Boosting)

- Extreme Gradient Boosting is an advanced implementation of the Gradient Boosting. This algorithm has high predictive power and is ten times faster than any other gradient boosting techniques. Moreover, includes the use of regularization which reduces overfitting and improves overall performance.
- Execution Speed: Generally, XGBoost is fast. Really fast when compared to other implementations of gradient boosting.
- Model Performance: XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems. The evidence is that it is the algorithm more used for competition winners on the Kaggle competitive data science platform.

# XGBoosting (Extreme Gradient Boosting)

- What is the difference between the gradient boosting machine) and xgboost (extreme gradient boosting)?
- Both xgboost and gbm follows the principle of gradient boosting. There are however, the difference in modeling details. Specifically, xgboost used a more regularized model formalization to control over-fitting, which gives it better performance.
- Objective Function : Training Loss + Regularization
- The regularization term controls the complexity of the model, which helps us to avoid overfitting.