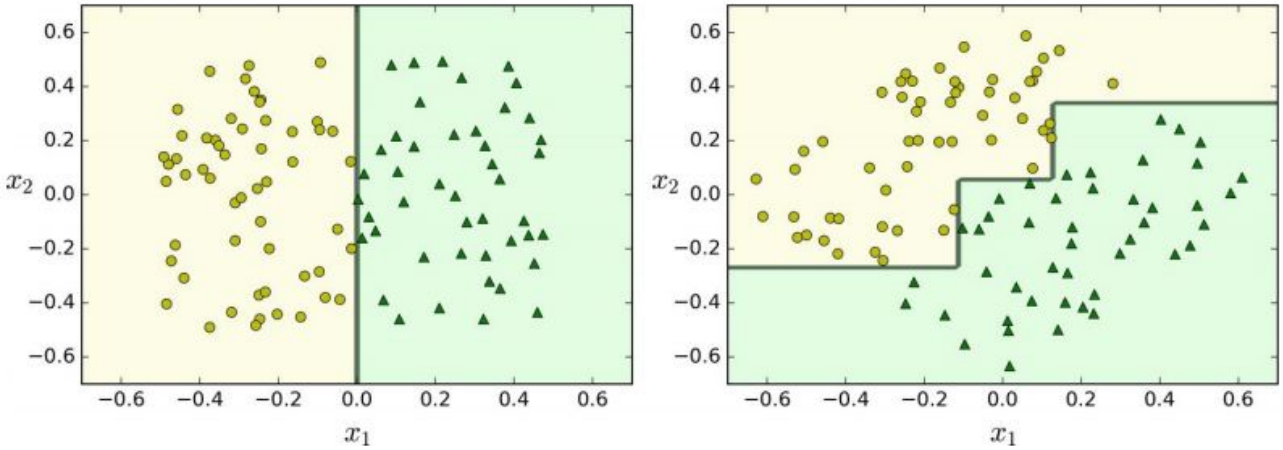
UNIVERSITÀ
DI PARMA
DIPARTIMENTO DI INGEGNERIA E ARCHITETTURA

Ensemble Learning

Gianfranco Lombardo, Ph.D
gianfranco.lombardo@unipr.it

Instability

- A decision tree often finds orthogonal decision boundaries
- Orthogonality makes them sensitive to training set rotation
- A decision tree can fits perfectly a training set but probably will not generalize well in some cases
- An idea to solve this problem is to apply the Principal Component Analysis (PCA) to data in order to have a better orientation of the training data

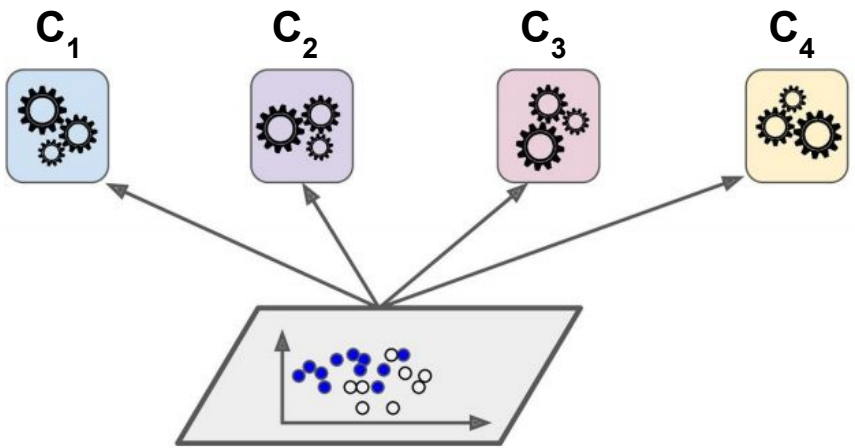


Ensemble learning

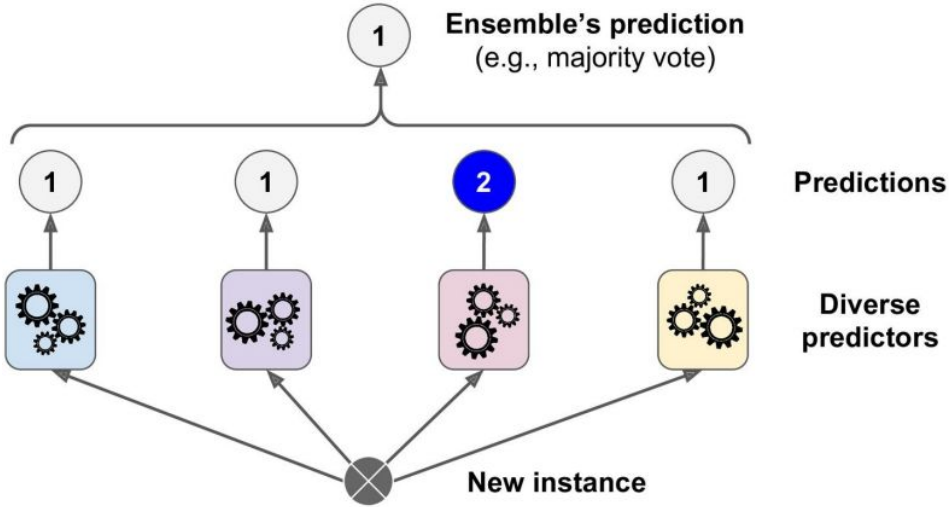
- When you are looking for an answer to a complex problem it is better to ask to different experts and then aggregate their answer (Wisdom of the crowd)

- Key idea: aggregate the predictions of a group of predictors to get better predictions than the best single predictor

Ensemble learning



Training



Test

Why the aggregation is better?

- Suppose you have a slightly biased coin: 51% chance of coming up heads
  - If you toss the coin 1000 times: you will have more or less 510 heads and 490 tails
  - What is the probability of having a majority of heads after 1000 tosses?
    - It is around 73%
    - If 10’000 tosses: the probability is around 95%
    - The more you toss the coin, the higher the probability is

- This is true, when each toss is **independent**
  - The result of each toss does not depend on the others!

Why the aggregation is better?

- N = 1000
- k = n° of heads that in N tosses has to be at least higher than (N/2)+1
- The probability that after N tosses the majority are heads is the sum of the probability of k successes with k in range of (N/2)+1 to N

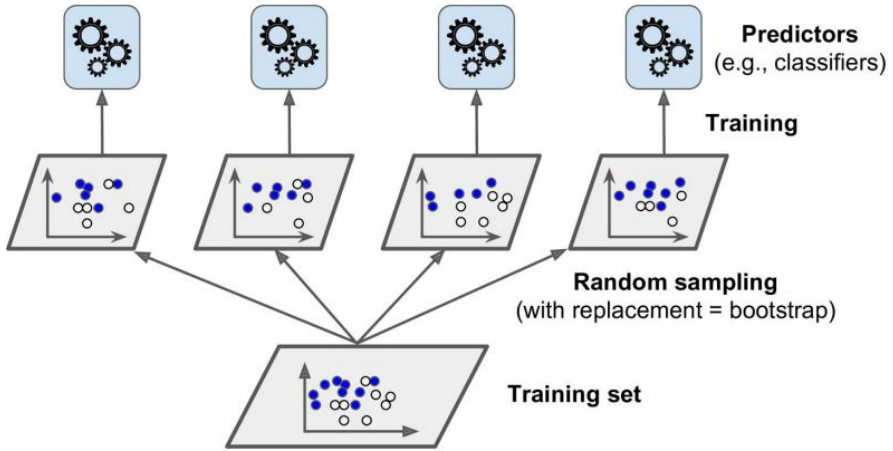
$$\sum_{k=(N/2)+1}^{N} \binom{N}{k} p^k (1-p)^{n-k}$$

Independent predictors

- Similarly, we can build 1000 classifiers that are individually correct only 51% of the time (slightly better than the random guessing -> weak classifier)
- If you predict the majority voted class, you can hope to get 73% accuracy
- This is true if **all classifiers are perfectly independent** !
    - They should make uncorrelated errors, otherwise they are likely to make the same type of errors and the majority could be wrong!
    - If all classifiers are trained on the same dataset, are they independent?

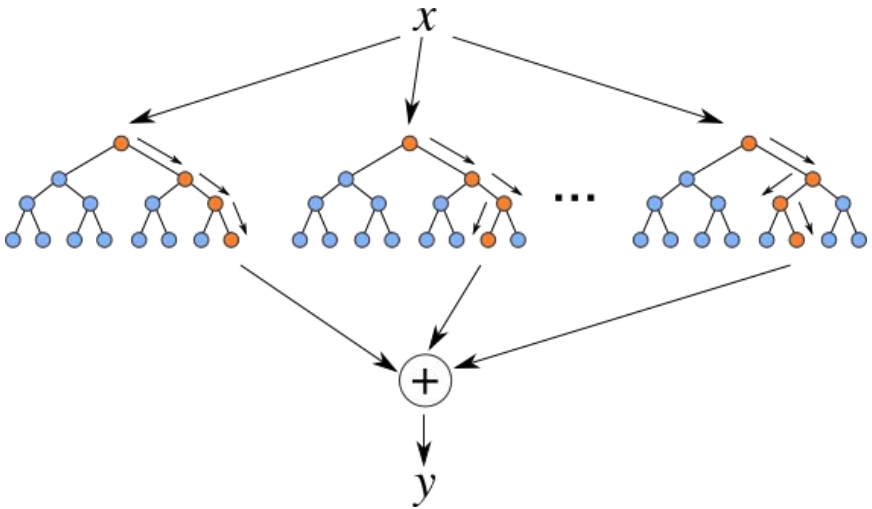
Bootstrap aggregating (Bagging)

- We have only one training-set and we want to use only one algorithm
- We can train the classifiers on different random subsets of the training-set
- When sampling is performed with replacement, this method is called **bootstrap**
- Each individual predictor has a higher bias than if it were trained on the original dataset, but aggregation reduces both bias and variance



Random forest

- Random forest is a bagging-based model where we make an aggregation of decision trees
- It also sub-samples a fraction of the features when fitting a decision tree to each bootstrap sample



Hard voting classifier

- Another solution to get independent classifiers is to use different algorithms
- Then we can make hard voting

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.ensemble import VotingClassifier

# import data
X, y = datasets.load_breast_cancer(return_X_y=True)
```

Hard voting classifier

```
log_clf = LogisticRegression(solver=liblinear)
rf_clf= RandomForestClassifier(n_estimators = 10)
svm_clf = SVC()

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.33, stratify=y)

voting_clf=VotingClassifier(estimators=[('lr',log_clf),('rf',rf_
clf),('svc',svm_clf)], voting='hard')
voting_clf.fit(X_train,y_train)
y_pred= voting_clf.predict(X_test)
print("Voting "+str(accuracy_score(y_test,y_pred)))
```

Hard voting classifier

```
log_clf.fit(X_train,y_train)
y_pred= log_clf.predict(X_test)
print("Logistic "+str(accuracy_score(y_test,y_pred)))

rf_clf.fit(X_train,y_train)
y_pred = rf_clf.predict(X_test)
print("Random forest "+str(accuracy_score(y_test,y_pred)))

svm_clf.fit(X_train,y_train)
y_pred = svm_clf.predict(X_test)
print("Support Vector Machine
"+str(accuracy_score(y_test,y_pred)))
```

Bagging with sklearn

from sklearn.ensemble import BaggingClassifier

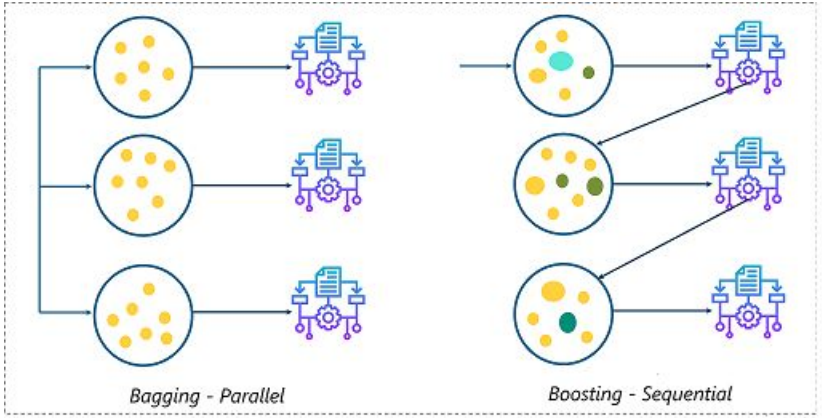
```
from sklearn.ensemble import BaggingClassifier
model= BaggingClassifier(n_estimators = 200)
model.fit(X_train,y_train)
```

```
from sklearn.ensemble import RandomForestClassifier
model= RandomForestClassifier(n_estimators = 1000)
model.fit(X_train,y_train)
```

BREAK

Boosting

- Similar to bagging, but now the idea is to train predictors sequentially
- Each predictor tries to correct its predecessor
- Many boosting methods available:
  - Adaptive boosting (AdaBoost)
  - Gradient boosting
  - XGboost (Extreme gradient boosting)



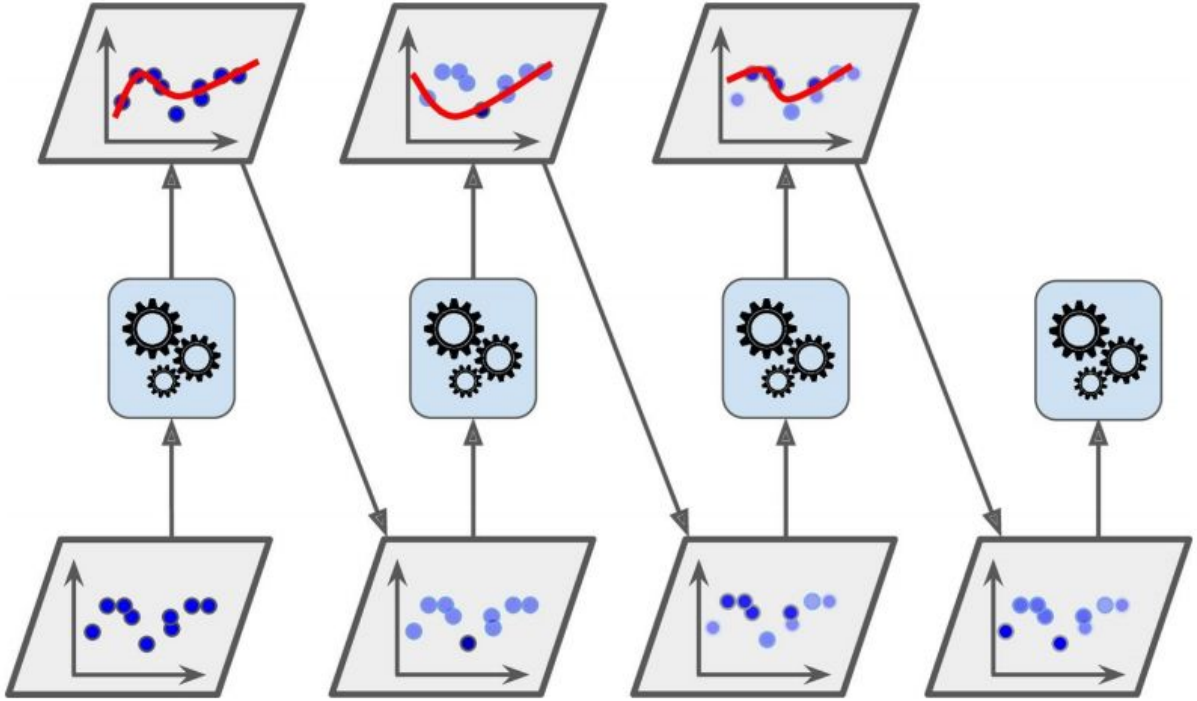
AdaBoost

- To correct a predecessor one way is to pay attention to the training instances that the predecessor under-fitted

- The new predictor will focus more on the hard cases (adaptive)
  a. A first base predictor is trained and used to make predictions on the training-set
  b. The relative weight of misclassified training instances is then increased
  c. A second classifier is trained using the updated weights and so on...

  d. A weight is assigned to each classifier
  e. Once all predictors are trained, the ensemble makes predictions like bagging, except that predictors have different weights depending on their accuracy on the training-set
-

AdaBoost



AdaBoost algorithm

- Each instance weight $\mathbf{w^{(i)}}$ is initially set to 1/m where m is the number of observations
- For each predictor **j** is computed a weighted error-rate $\mathbf{r_j}$

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

where y-hat is the j^{th} predictors' prediction for the i^{th} instance

AdaBoost algorithm

- A predictor's weight α_j is then computed using a learning-rate parameter **η**
- The more accurate the predictor is, the higher its weight will be.

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

- The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative

AdaBoost algorithm

- After that, the instance weights are updated using the equation below

- All the instance weights are then normalized dividing by the sum of all weights

$$\text{for } i = 1, 2, \cdots, m$$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

AdaBoost predictions

- Finally, AdaBoost computes the predictions of all predictors and weights them using α_j
- The predicted class is the one that gets the majority of weighted votes

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x})=k}}^{N} \alpha_j$$

where N is the number of predictors

Gradient Boosting

- The key idea of GB is additive modelling
  - We can have a final learner $F_L = f_0(x)+f_1(x)+f_2(x)+...+f_{L-1}(x)$
  - Where L is the number of learners we want to use (for example n°of trees)
  - **$f_0(x)$** is the first predictor we train on the training-set
  - Each epoch of training we add a new learner that tries to increase the accuracy of the previous one by minimizing a Loss function using gradient descent and not by increasing instances' weight like AdaBoost

Gradient Boosting for Regression (Classification)

- Let's play a game: You are given a dataset in the form of $(x_1,y_1),...,(x_n,y_n)$.
- The task is to fit a model F(x) to minimize the square loss (sum of square difference between y_true and y_pred)
- Suppose your friend wants to help you and gives you his model F
- You check his model and find the model is good but not perfect because it makes some mistakes:
  - For example: $F(x_1) = 0.5$ while $y_1 = 0.6$
  - $F(x_2) = 1.8$ while $y_2 = 1.7$
- You cannot change the model F or change its parameters but you want to improve it

Gradient Boosting for Regression (Classification)

- We can add a model (regression tree for example) **h** to F, so the new prediction will be F(x)+**h**(x)
- You wish to improve the model such that
  - $F(x_1)+h(x_1) = y_1$
  - $F(x_2)+h(x_2) = y_2$
  - $F(x_n)+h(x_n) = y_n$

  - $\mathbf{h}(x_1) = y_1 - F(x_1)$
  - $\mathbf{h}(x_2) = y_2 - F(x_2)$
  - $\mathbf{h}(x_n) = y_n - F(x_n)$

Can any regression tree h achieve this goal perfectly ? Probably not! But some regression might be able to do this approximately

Gradient Boosting for Regression (Classification)

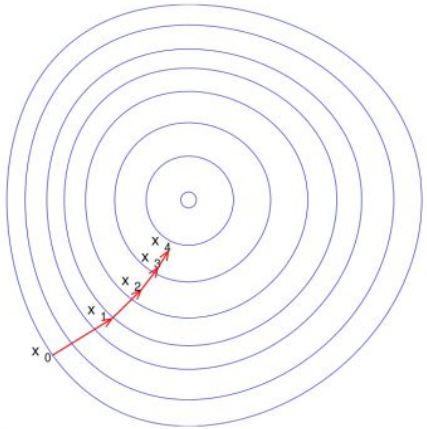
- To get this predictor **h** we can fit a regression tree with a different formulation of the original dataset where the target value to predict will be *y - F(x)*
- $\{x_1,y_1 - F(x_1),...,(x_n,y_n - F(x_n)\}$
- $y_i - F(x_i)$ are usually called residuals
- We want to minimize the entire loss for all training examples (1,...,n)
- To minimize this loss we can use the gradient descent

Reminder: Gradient descent

Gradient Descent

Minimize a function by moving in the opposite direction of the gradient.

$$\theta_i := \theta_i - \rho \frac{\partial J}{\partial \theta_i}$$



Gradient Boosting for Regression (Classification)

- Loss function $L(y, F(x)) = (y - F(x))^2 / 2$
- We want to minimize $J = \sum_i L(y_i, F(x_i))$ by adjusting $F(x_1), F(x_2) \ldots F(x_n)$

$$\frac{\partial J}{\partial F(x_i)} = \frac{\partial \sum_i L(y_i, F(x_i))}{\partial F(x_i)} = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = F(x_i) - y_i$$

- So we can interpret residuals as negative gradients

$$y_i - F(x_i) = -\frac{\partial J}{\partial F(x_i)}$$

Gradient Boosting for Regression (Classification)

- So at the end we can go back to see the predictors at stage m+1 and define it as

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

$$F_{m+1}(x) = F_m(x) - \nabla_m L(y, F_m(x)) = F_m(x) - \left[\frac{\partial L(y, F_m(x))}{\partial F_m(x)}\right]$$

- With a learning-rate equal to 1

Boosting with sklearn

```
from sklearn.ensemble import AdaBoostClassifier
model= AdaBoostClassifier(n_estimators = 200)
model.fit(X_train,y_train)
```

```
from sklearn.ensemble import GradientBoostingClassifier
model= GradientBoostingClassifier(n_estimators = 200)
model.fit(X_train,y_train)
```

```
from xgboost import XGBClassifier
model = XGBClassifier(n_estimators = 200)
model.fit(X_train, y_train)
```