

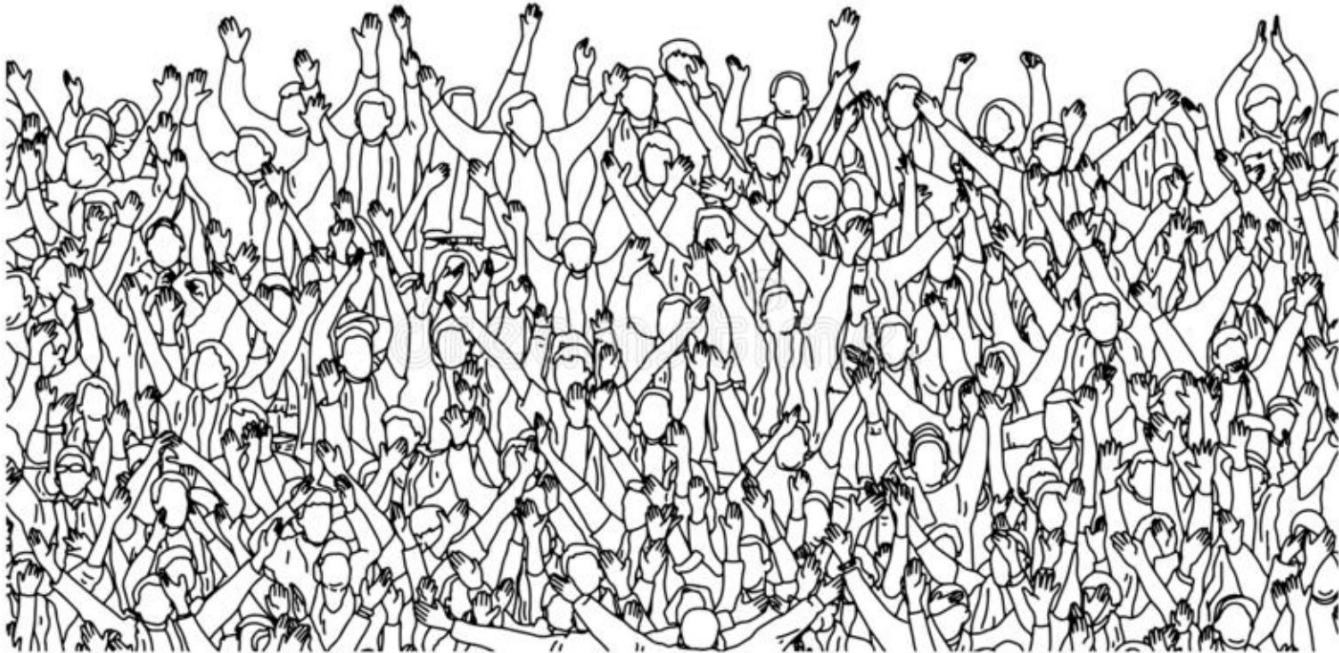
Métodos de Ensamble



**UNIVERSIDAD
CATÓLICA**
DE CÓRDOBA
JESUITAS

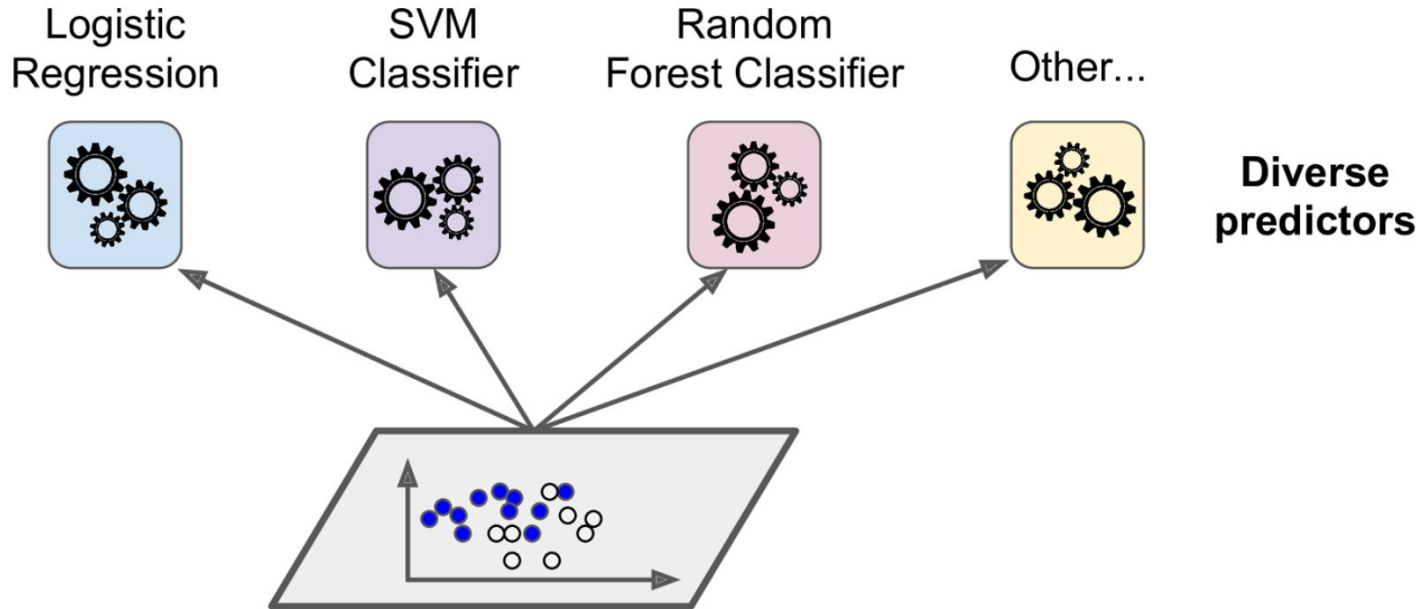
Dr. Francisco Arduh
2023

¿Cuál es la idea de los métodos de ensamble?



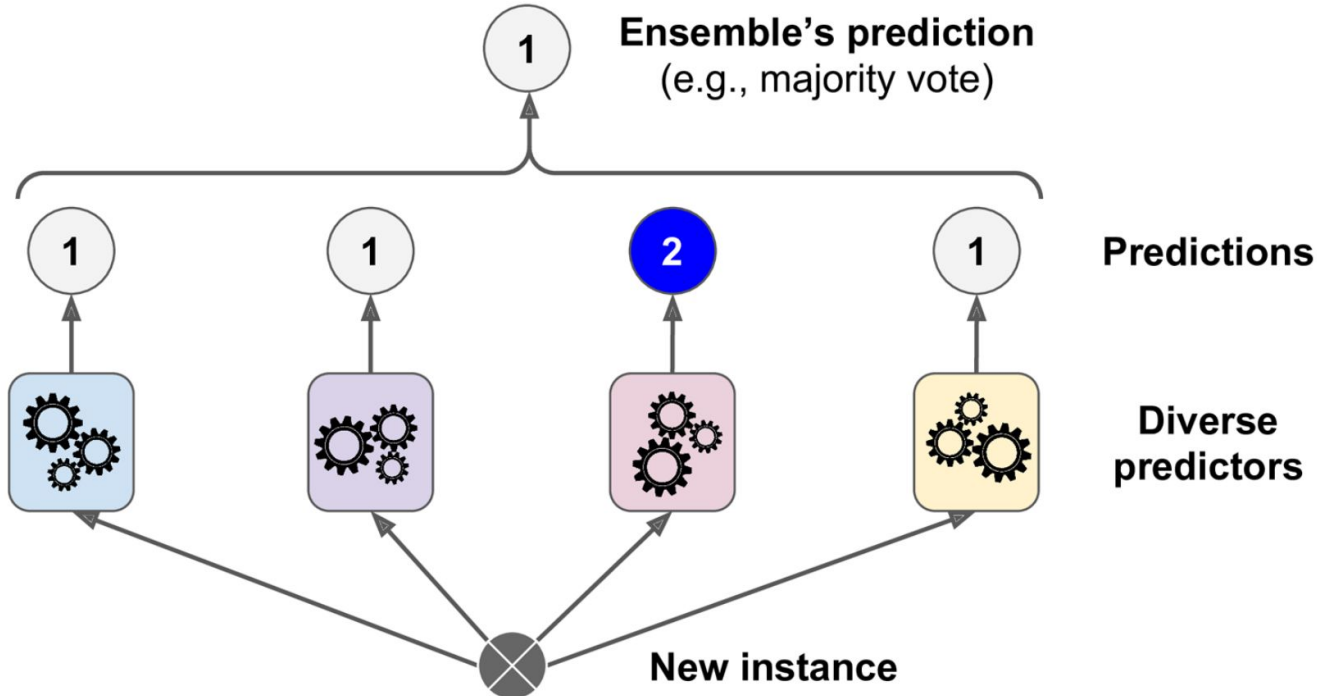
Votación por mayoría

Por ej: Se cuenta con varios clasificadores entrenados con un accuracy del 80% ...



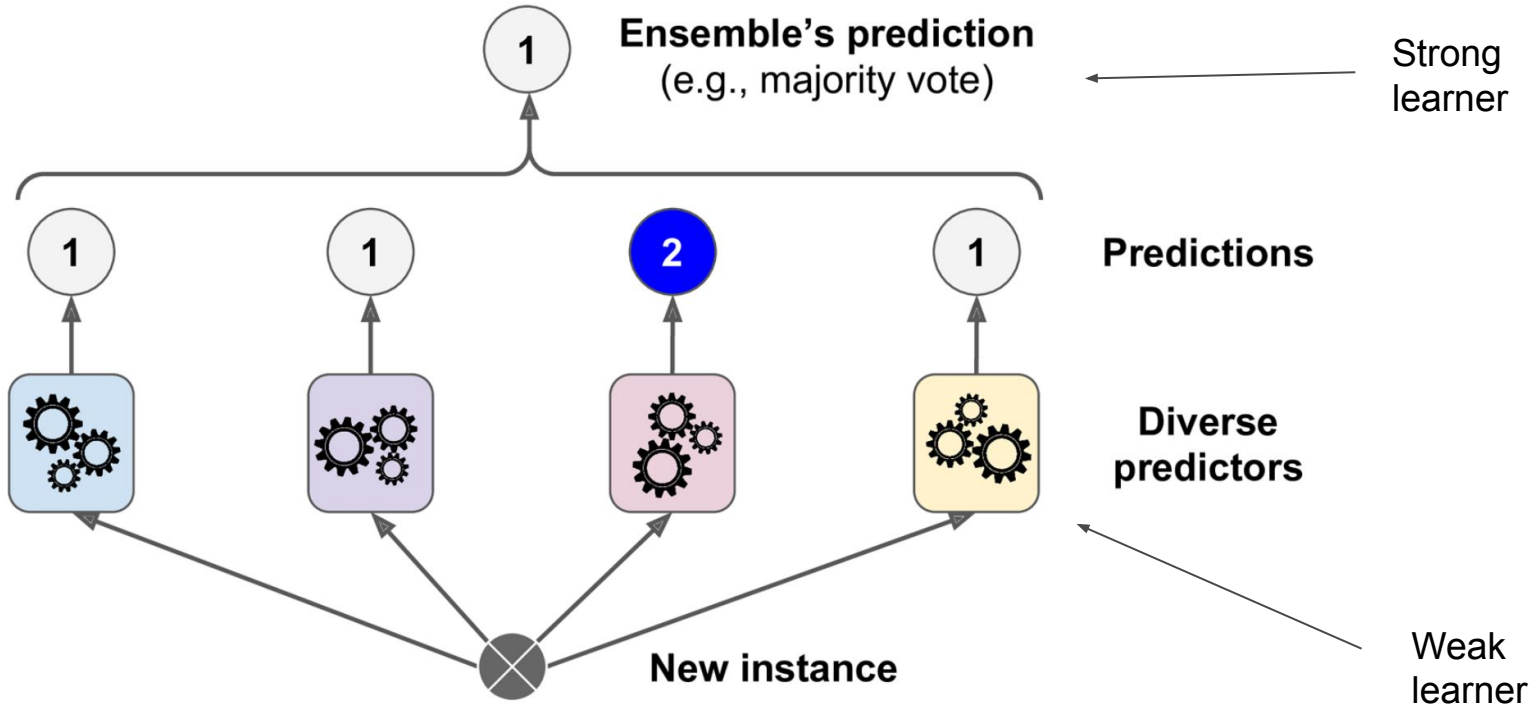
Votación por mayoría

A partir de estos clasificadores, se construye uno nuevo que toma la clase más votada como predicción (**hard voting classifier**)



Votación por mayoría

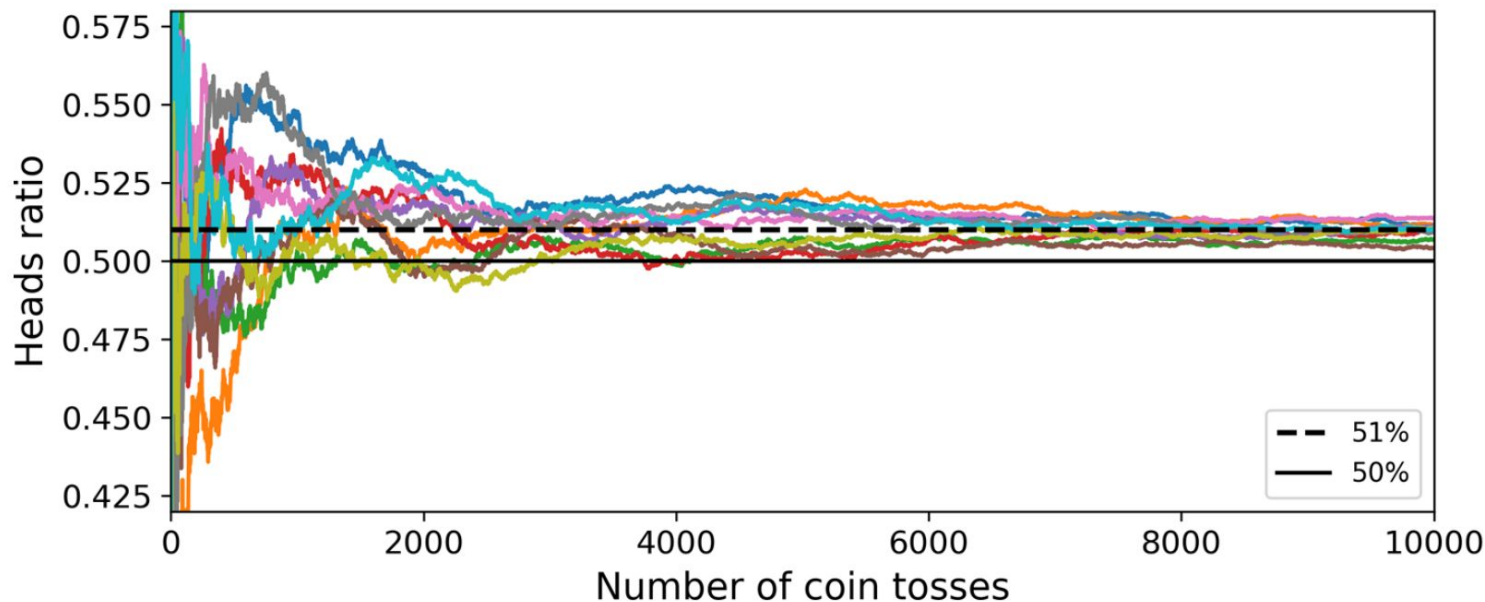
A partir de estos clasificadores, se construye uno nuevo que toma la clase más votada como predicción (**hard voting classifier**)



¿Por qué funciona esto?

Supongamos que tenemos un moneda con bias: $P(\text{cara})=51\%$, $P(\text{cruz})=49\%$

- 1000 lanzamientos $\Rightarrow P(\#\text{cara} > \#\text{cruz}) = 75\%$
- 10000 lanzamientos $\Rightarrow P(\#\text{cara} > \#\text{cruz}) = 97\%$

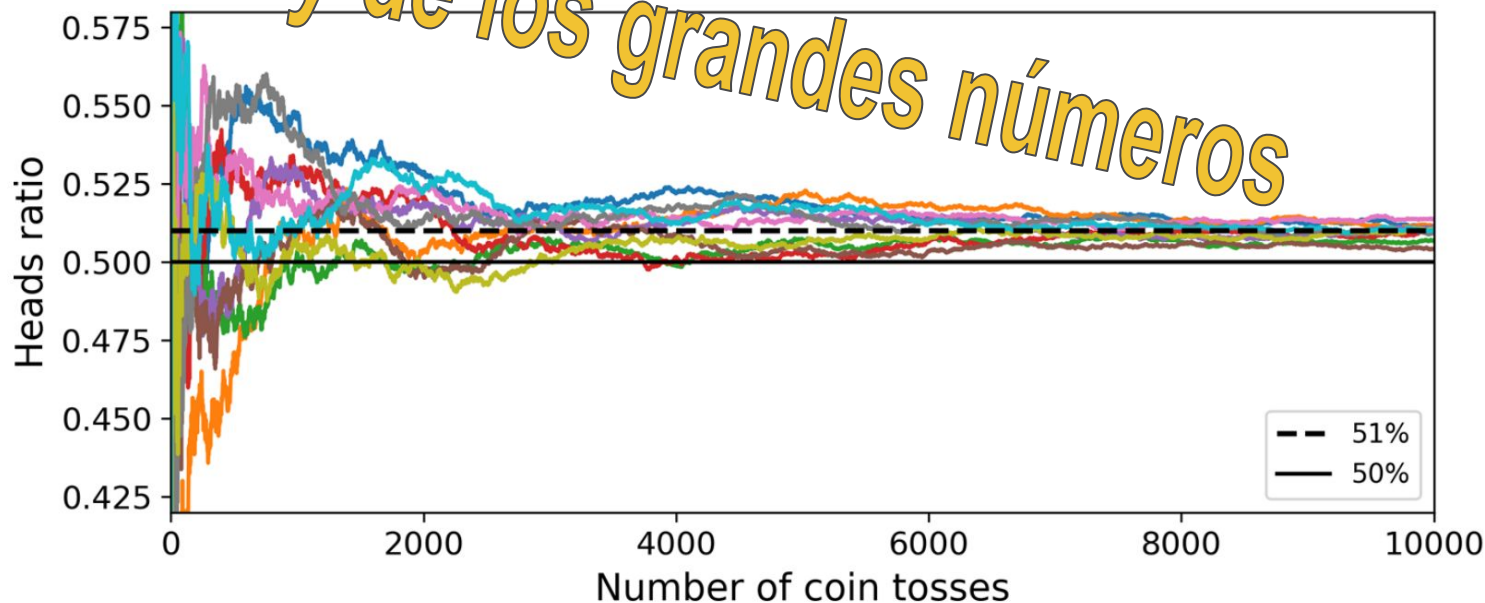


¿Por qué funciona esto?

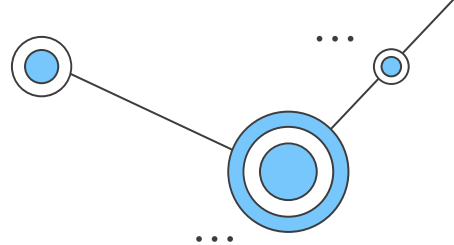
Supongamos que tenemos un moneda con bias: $P(\text{cara})=51\%$, $P(\text{cruz})=49\%$

- 1000 lanzamientos $\Rightarrow P(\#\text{cara} > \#\text{cruz}) = 75\%$
- 10 000 lanzamientos $\Rightarrow P(\#\text{cara} > \#\text{cruz}) = 97\%$

Ley de los grandes números

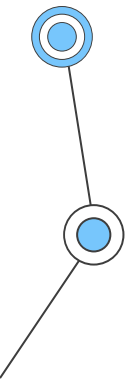


¿Por qué funciona esto?

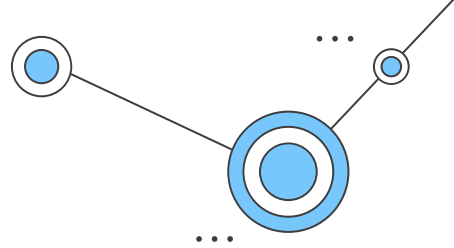


La misma lógica funciona con los clasificadores.

- Si construimos 1000 clasificadores con un accuracy del 51%, tomando el voto de la mayoría llegaríamos al 75%.
- Lo anterior es verdad solo si los clasificadores son independientes.
- Una forma de tratar que sean lo más independientes posibles es utilizar distintos tipos de clasificadores, los cuales van a cometer distintos tipos de errores.



Votación por mayoría: en Scikit-Learn



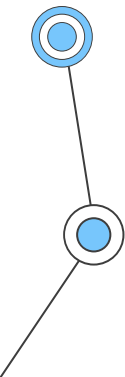
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Utilizando el test set:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```



Votación por mayoría: en Scikit-Learn

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
```

```
log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()
```

```
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

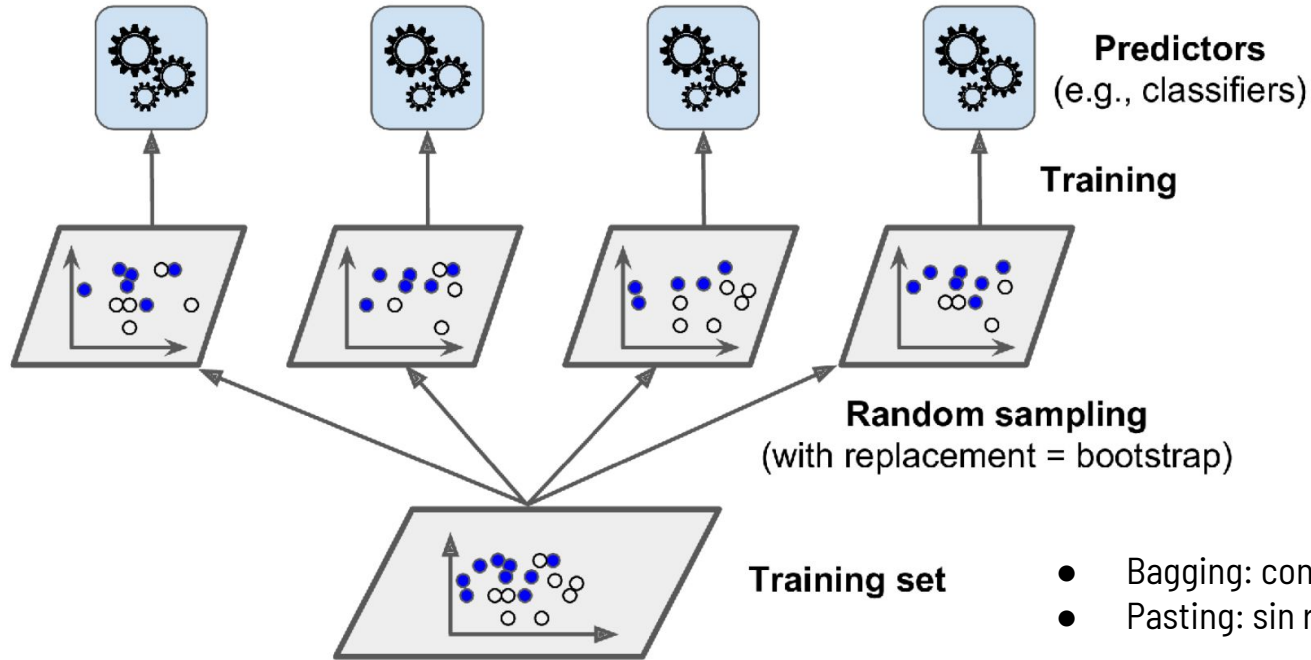
Es posible utilizar soft voting, cambiando esta variable por 'soft'

Utilizando el test set:

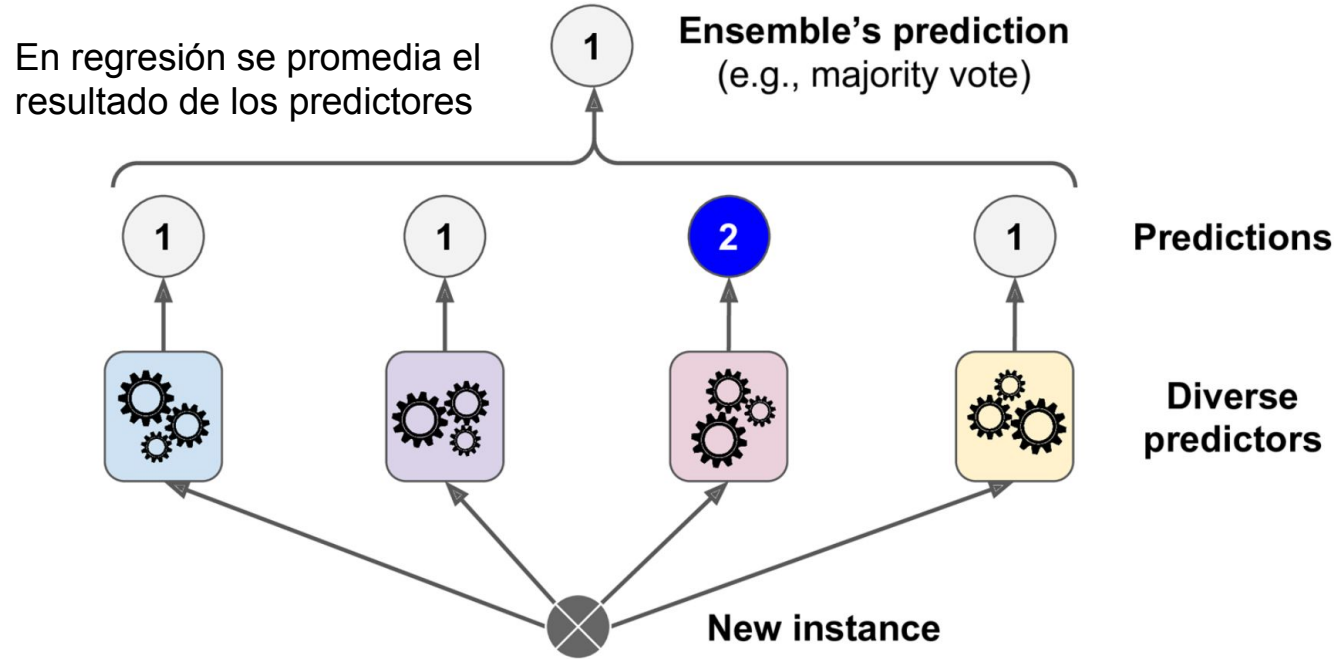
```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

Soft voting: elige la clase con probabilidad más alta entre todos los clasificadores individuales

Bagging y Pasting



Bagging y Pasting



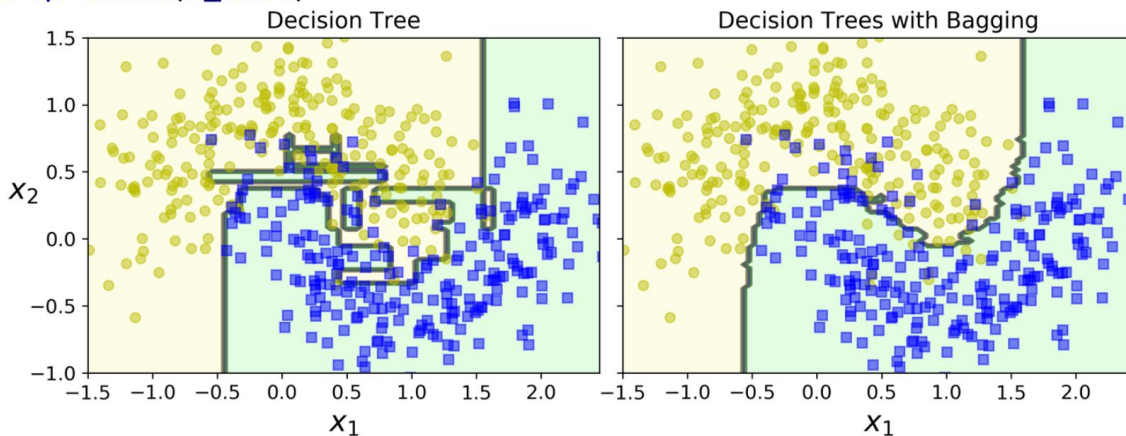
Nota: cada predictor tiene un mayor bias que si se hubiese entrenado con el dataset original, pero la combinación de todos los predictores tiene menor bias y varianza.

Bagging y Pasting: en Scikit-learn

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
```

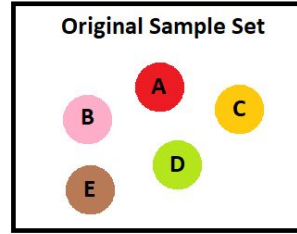
```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

- Pasting: bootstrap='False'
- Realiza soft voting siempre que los predictores contenga el método **predict_proba()**



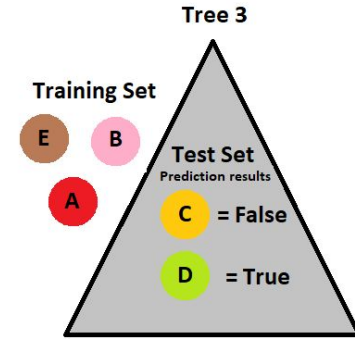
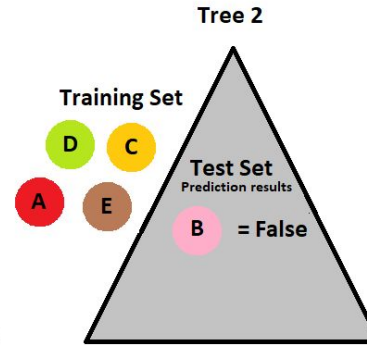
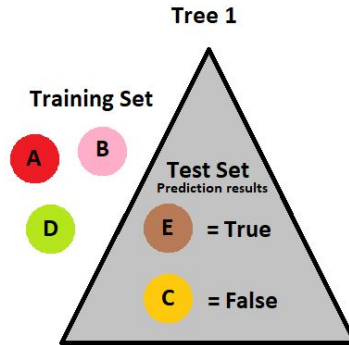
Evaluación Out-of-Bag

- En promedio cada predictor utiliza un ~63.2% de la muestra.
- El ~36.8% restante qué no se utiliza se denomina out-of-bag.

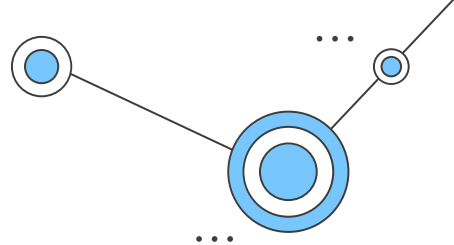


Samples	Majority Vote
A	True
B	False
C	False
D	True
E	True

⇒ 2/5 Out of Bag Error



Evaluación Out-of-Bag: en Scikit-learn



Con `oob_score`, obtengo la accuracy en out-of-bag set:

```
>>> bag_clf = BaggingClassifier(  
...     DecisionTreeClassifier(), n_estimators=500,  
...     bootstrap=True, n_jobs=-1, oob_score=True)  
...  
>>> bag_clf.fit(X_train, y_train)  
>>> bag_clf.oob_score_  
0.9013333333333332
```

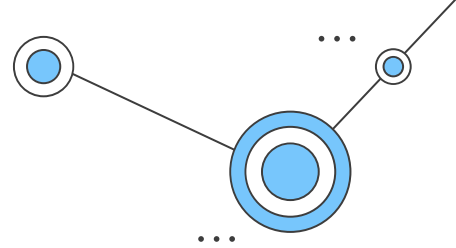
Comparo con lo que obtengo en el test set:

```
>>> from sklearn.metrics import accuracy_score  
>>> y_pred = bag_clf.predict(X_test)  
>>> accuracy_score(y_test, y_pred)  
0.91200000000000003
```

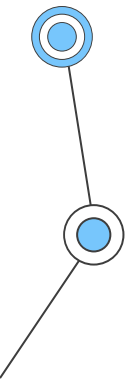
También puedo obtener el valor de la función de decisión:

```
>>> bag_clf.oob_decision_function_  
array([[0.31746032, 0.68253968],  
       [0.34117647, 0.65882353],  
       [1.          , 0.          ],  
       ...  
       [1.          , 0.          ],  
       [0.03108808, 0.96891192],  
       [0.57291667, 0.42708333]])
```

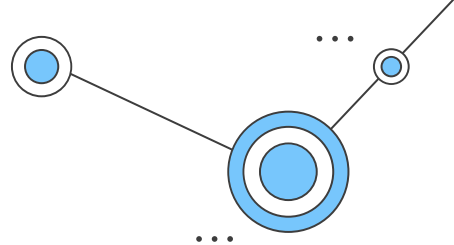
Random Patches y Random Subspaces



- Con `max_features` y `bootstrap_feature` puedo controlar la utilización de las características. Esto es útil cuando se trabaja con muchas características.
- Variando la utilización de las características y las instancias se denomina Random Patches.
- Variando la utilización sólo de las características se denomina Random Subspaces.
- Variando la utilización de las características se cambia un poco de bias por disminución de la varianza



Random Forest



Random Forest no es otra cosa que utilizar el método de bagging en Decision Trees.

Utilizar:

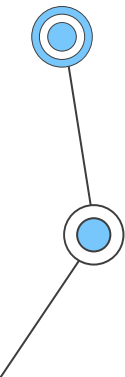
```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

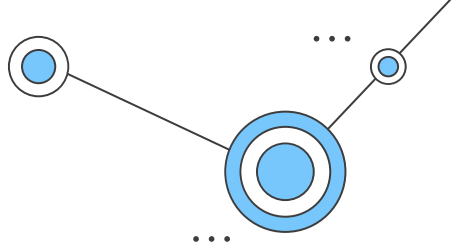
y_pred_rf = rnd_clf.predict(X_test)
```

Es equivalente a:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```



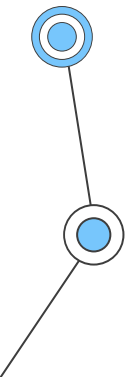
Extra-Trees



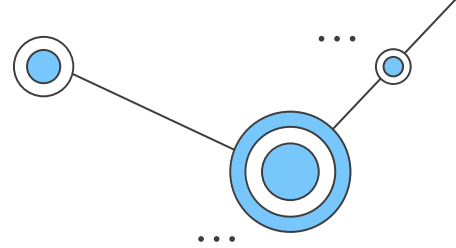
Extra-Trees (Extremete Randomized Trees)

- Utilizan selecciones aleatorias en las características (No optimiza).
- Son más rápidos de entrenar que RandomForest

```
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = ExtraTreesClassifier(n_estimators=100, random_state=0)
>>> clf.fit(X, y)
ExtraTreesClassifier(random_state=0)
>>> clf.predict([[0, 0, 0, 0]])
array([1])
```



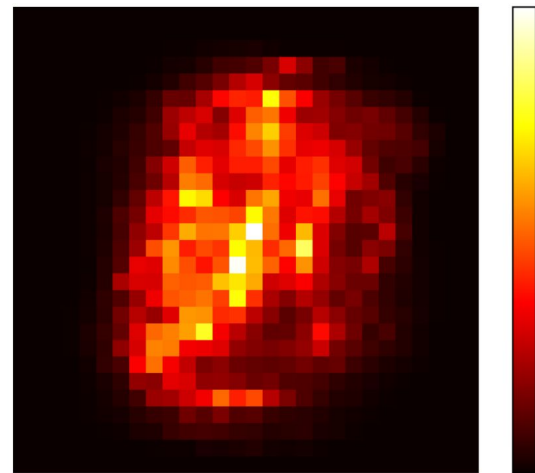
Random Forest: importancia de características



Random Forest computa la importancia de la característica en función de cuanto se reduce la impureza Gini (en promedio en todo los DT).

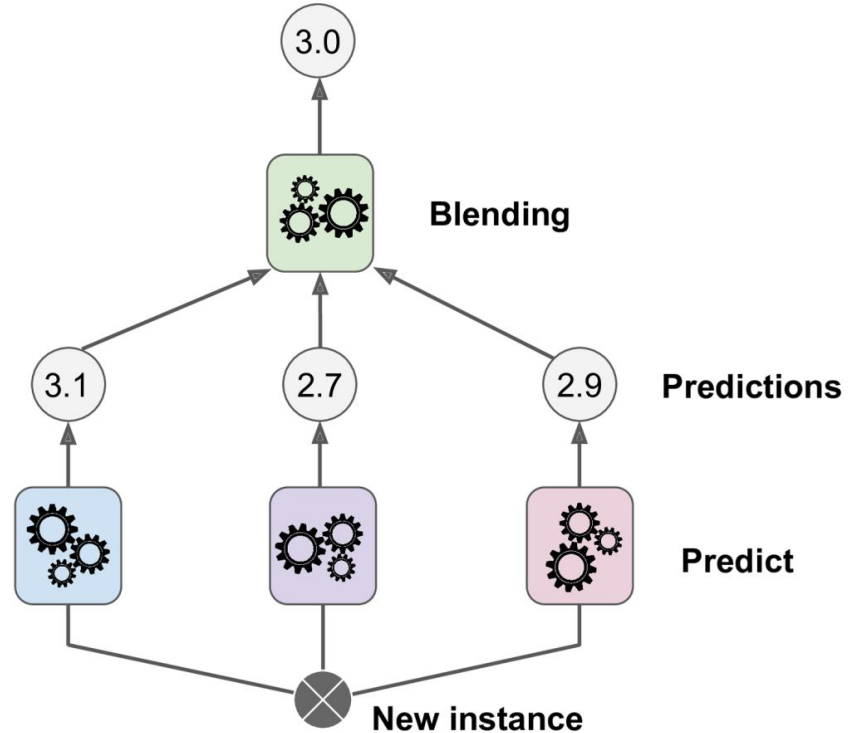
```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Importancia de pixel en MNIST:



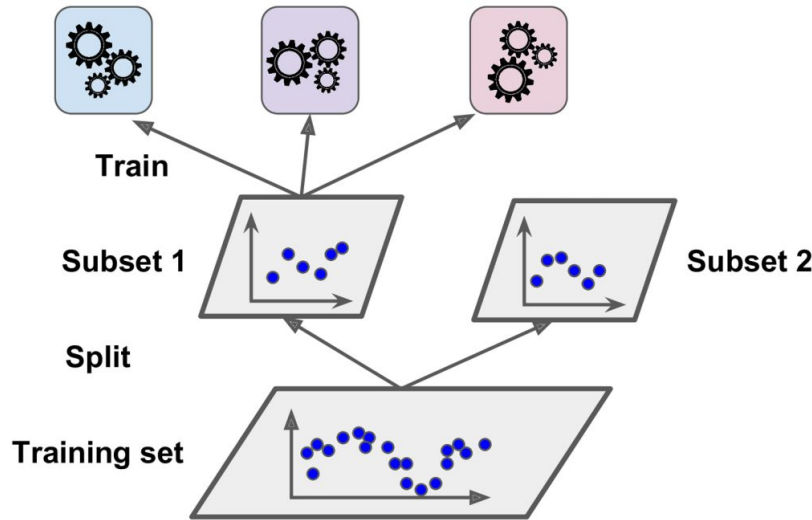
Stacking

- En vez de utilizar hard voting (o soft) para juntar las predicciones, se entrena un modelo para realizar esta tarea.
- Al predictor que mezcla las observaciones se lo denomina *blender* o *meta learner*

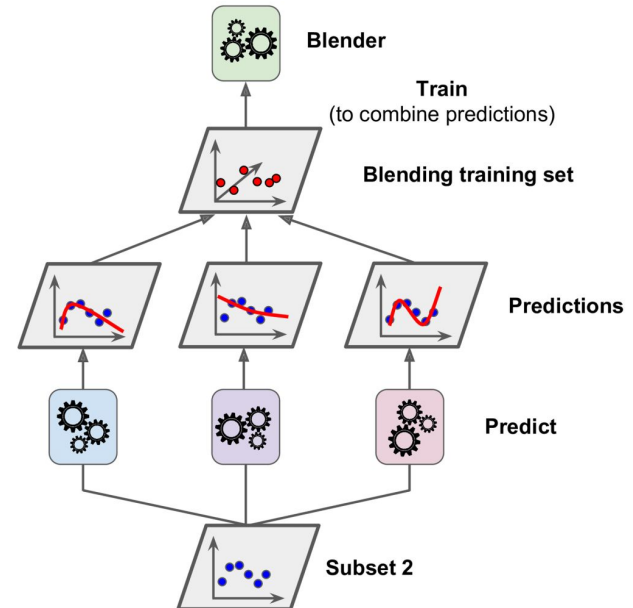


Stacking: ¿cómo se entrenan?

Paso 1: Dividir la muestra en dos
con la primer parte entrenar los
primeros predictores

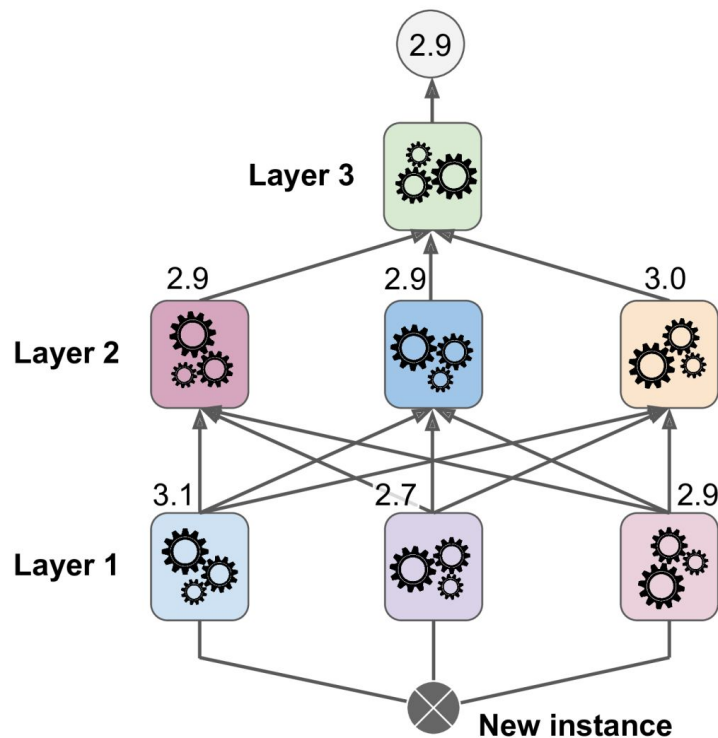


Paso 2: Con la segunda muestra,
se realiza la predicción y se
entrena el blender



Stacking: ¿cómo se entrenan?

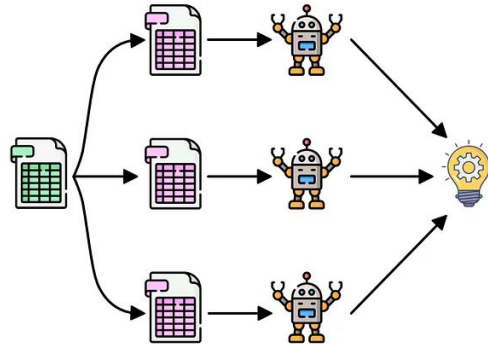
- Es posible utilizar varios blenders.
- En este caso deberíamos dividir la muestra en 3.



Boosting

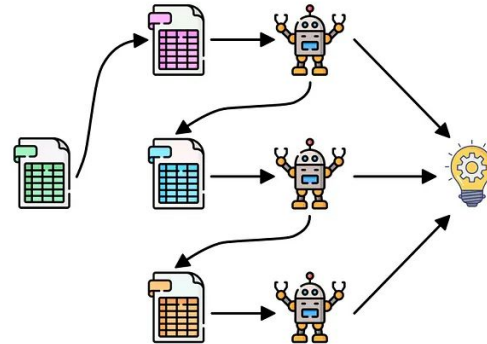
- Se entrenan predictores secuencialmente donde cada modelo intenta arreglar los errores de los modelos anteriores.

Bagging



Parallel

Boosting

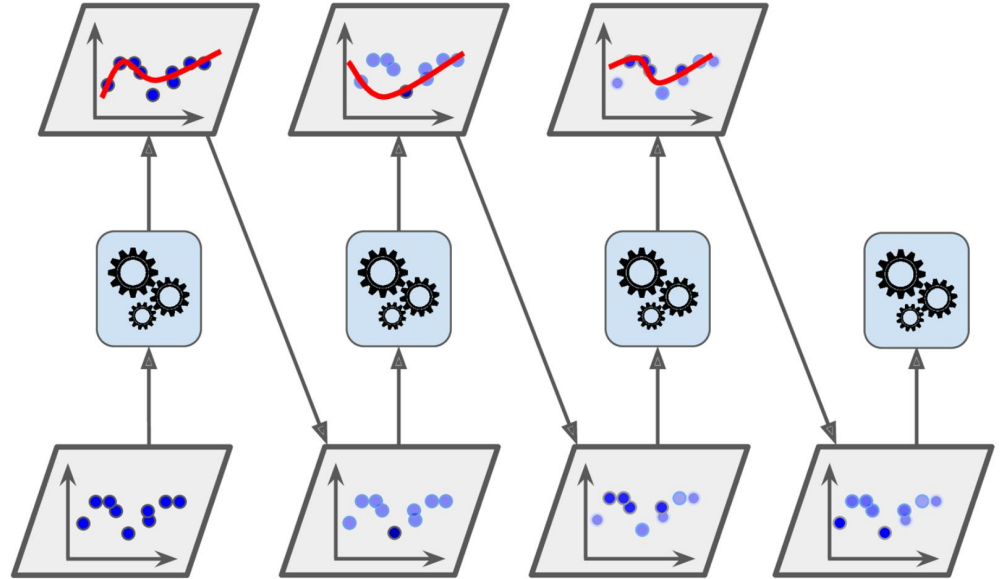


Sequential

- Los algoritmos secuenciales no se pueden paralelizar.

AdaBoost (Adaptative Boost)

- Una forma de corregir al predictor predecesor es darle más peso a los casos subajustados.
- Como weak learner se utilizan Decision Stump (DT con $\text{max_depth}=1$).



AdaBoost: ¿Cómo funciona?

1. Inicialmente se pesa cada instancia $1/m$, siendo m el número de instancias.
2. Se entrena un weak learner teniendo en cuenta el peso de las muestras.
3. Se calcula el error cometido por el predictor. Se toma en el caso de clasificación la tasa de instancias mal identificadas.
4. Se calcula el peso que se le va a asignar al predictor, dado por:

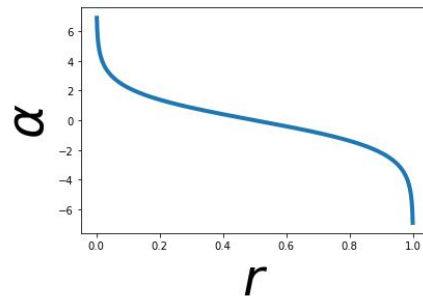
$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Siendo r_j el error cometido por el predictor j y η un learning rate.

5. Luego se actualizan los pesos siguiendo esta regla:

$$\begin{aligned} &\text{for } i = 1, 2, \dots, m \\ &w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases} \end{aligned}$$

6. Se normalizan los pesos (dividiendo por la suma de los nuevos pesos)
7. Se repite del punto 2 al 6 hasta alcanzar el número de estimadores deseados.
8. Finalmente para realizar predicciones se utilizan todos los estimadores pesado por α_j



AdaBoost: En Scikit-learn

- Scikit-learn utiliza una versión multiclase del algoritmo AdaBoost llamada ... SAMME.
- SAMME.R es un variante de SAMME qué utiliza probabilidades

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.ensemble import AdaBoostClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                           n_informative=2, n_redundant=0,
...                           random_state=0, shuffle=False)
>>> clf = AdaBoostClassifier(
...     DecisionTreeClassifier(max_depth=1), n_estimators=200,
...     algorithm="SAMME.R", learning_rate=0.5)
>>> clf.fit(X, y)
AdaBoostClassifier(n_estimators=100, random_state=0)
>>> clf.predict([[0, 0, 0, 0]])
array([1])
>>> clf.score(X, y)
0.977...
```