Clase12 IMA539

Alejandro Ferreira Vergara

April 28, 2023

1 Aprendizaje en Conjunto

1.1 ¿Qué aprenderemos hoy?

- **Clasificación combinada mediante votación simple**
- **AdaBoost**
- **Implementación con Python**

2 Combinar Modelos mediante Votación

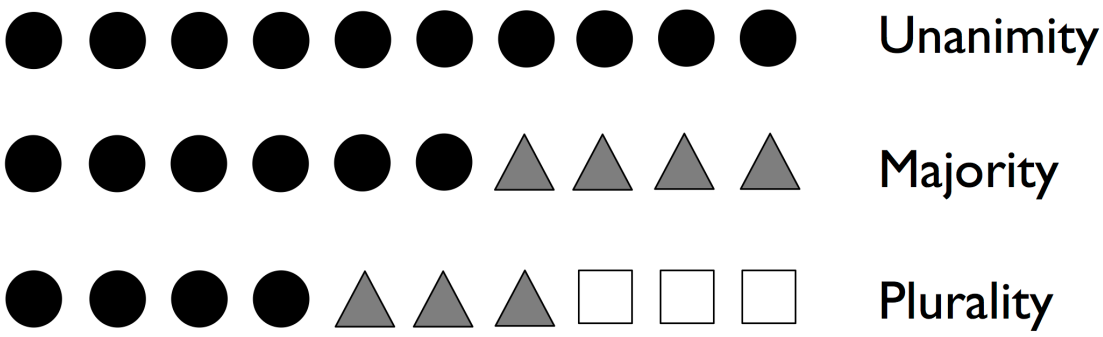
El objetivo de los métodos de Aprendizaje en Conjunto es combinar diferentes clasificadores en un metaclasificador que tenga un mejor rendimiento de generalización que cada clasificador individual por separado.

- **Votación por mayoría (problemas binarios):** Tomamos la etiqueta de clase predicha por la mayoría de los clasificadores.
- **Votación por pluralidad (problemas multiclase):** Seleccionamos la etiqueta de clase que ha recibido más votos (moda).

[1]:
```
from IPython.display import Image

Image(filename=r'clase12/12_1.png', width=500)
```

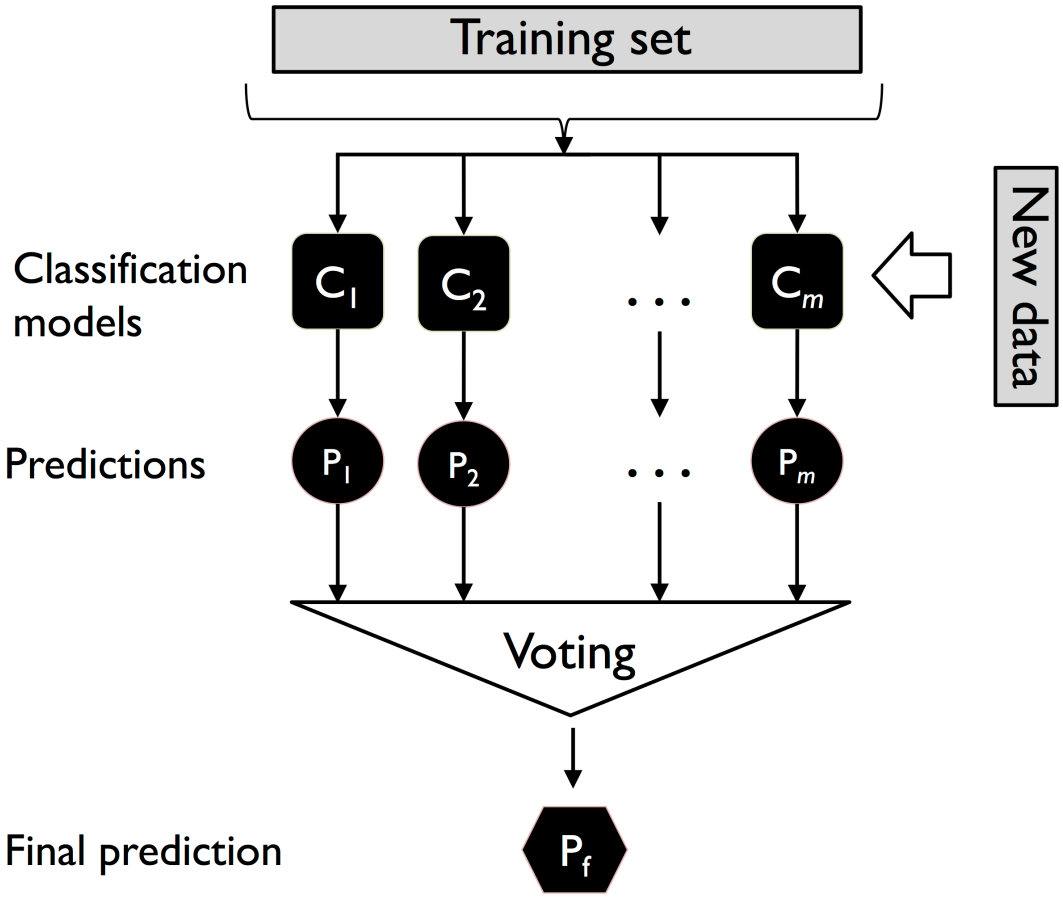
[1]:



Con el conjunto de entrenamiento ajustamos m clasificadores: $C_1, C_2, ..., C_m$.

[2]:
```
Image(filename=r'clase12/12_2.png', width=400)
```

[2]:



- **Predicción:** $\hat{y} = moda\left\{C_1(x), C_2(x), \cdots, C_m(x)\right\}$
- **Caso clasificación binaria:** $C(x) = \hat{y} = sign\left[\sum_j^m C_j(x)\right] = \begin{cases} 1 & \text{, si } \sum_i C_j(x) \geq 0 \\ -1 & \text{, en otro caso} \end{cases}$

¿Un clasificador por votación podría ser mejor que un clasificador individual?

Supongamos que para una tarea particular, los m clasificadores (binarios) base tienen la misma tasa de error, que llamaremos ε. Además, supongamos que los clasificadores son independientes y que las tasas de error no están correlacionadas. Bajo estos supuestos, podemos expresar la probabilidad de error de un conjunto de clasificadores base como una función de masa de probabilidad de una distribución binomial:

$$P(y \geq k) = \sum_{k=1}^{n} \binom{n}{k} \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{ensemble}$$

En otras palabras, estamos calculando la probabilidad de que la predicción del conjunto sea errónea.

Por ejemplo, pensemos en 11 clasificadores base ($m = 11$), en el que cada clasificador tiene una tasa de error de 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=1}^{11} \binom{11}{k} 0.25^k (1 - 0.25)^{11-k} = 0.034 = \varepsilon_{ensemble}$$

[ ]:
```
from scipy.special import comb
import math
import numpy as np
import matplotlib.pyplot as plt

def ensemble_error(n_classifier, error):
    k_start = int(math.ceil(n_classifier / 2.))
    probs = [comb(n_classifier, k) * error**k * (1-error)**(n_classifier - k)
             for k in range(k_start, n_classifier + 1)]
    return sum(probs)

error_range = np.arange(0.0, 1.01, 0.01)
ens_errors = [ensemble_error(n_classifier=11, error=error) for error in␣
 ↪error_range]

plt.plot(error_range,
         ens_errors,
         label='Ensemble error',
         linewidth=2)

plt.plot(error_range,
         error_range,
         linestyle='--',
         label='Base error',
         linewidth=2)

plt.xlabel('Base error')
plt.ylabel('Base/Ensemble error')
plt.legend(loc='upper left')
plt.grid(alpha=0.5)
#plt.savefig('images/07_03.png', dpi=300)
plt.show()
```

2.1 Implementación de un Clasificador por Mayoría Simple

- **Voto Mayoritario Ponderado (por mayoría):**

  $\hat{y} = arg\ \underset{i}{max} \sum_{j=1}^{m} w_j \chi_A \left(C_j(x) = i\right)$

  $\hat{y}$: clase predicha por el conjunto de clasificadores

w_j : peso asociado al clasificador base C_j

χ_A : función característica, $\left[C_j(x) = i \in A\right]$

A : conjunto de etiquetas de clase únicas.

- **Para pesos iguales, podemos simplificar la ecuación anterior y escribirla como:**

  $\hat{y} = moda\left\{C_1(x), C_2(x), \cdots, C_m(x)\right\}$

[ ]:
```
np.argmax(np.bincount([0, 0, 1], weights=[0.2, 0.2, 0.6]))
```

[ ]:
```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import operator

class MajorityVoteClassifier(BaseEstimator, ClassifierMixin):
    """ A majority vote ensemble classifier

    Parameters
    ----------
    classifiers : array-like, shape = [n_classifiers]
      Different classifiers for the ensemble

    vote : str, {'classlabel', 'probability'} (default='label')
      If 'classlabel' the prediction is based on the argmax of
        class labels. Else if 'probability', the argmax of
        the sum of probabilities is used to predict the class label
        (recommended for calibrated classifiers).

    weights : array-like, shape = [n_classifiers], optional (default=None)
      If a list of `int` or `float` values are provided, the classifiers
      are weighted by importance; Uses uniform weights if `weights=None`.

    """
    def __init__(self, classifiers, vote='classlabel', weights=None):

        self.classifiers = classifiers
        self.named_classifiers = {key: value for key, value in␣
 ↪_name_estimators(classifiers)}
        self.vote = vote
        self.weights = weights

    def fit(self, X, y):
        """ Fit classifiers.
```

```
        Parameters
        ----------
        X : {array-like, sparse matrix}, shape = [n_samples, n_features]
            Matrix of training samples.

        y : array-like, shape = [n_samples]
            Vector of target class labels.

        Returns
        -------
        self : object

        """
        if self.vote not in ('probability', 'classlabel'):
            raise ValueError("vote must be 'probability' or 'classlabel'"
                             "; got (vote=%r)"
                             % self.vote)

        if self.weights and len(self.weights) != len(self.classifiers):
            raise ValueError('Number of classifiers and weights must be equal'
                             '; got %d weights, %d classifiers'
                             % (len(self.weights), len(self.classifiers)))

        self.lablenc_ = LabelEncoder()
        self.lablenc_.fit(y)
        self.classes_ = self.lablenc_.classes_
        self.classifiers_ = []
        for clf in self.classifiers:
            fitted_clf = clone(clf).fit(X, self.lablenc_.transform(y))
            self.classifiers_.append(fitted_clf)
        return self

    def predict(self, X):
        """ Predict class labels for X.

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape = [n_samples, n_features]
            Matrix of training samples.

        Returns
        ----------
        maj_vote : array-like, shape = [n_samples]
            Predicted class labels.

        """
```

```
        if self.vote == 'probability':
            maj_vote = np.argmax(self.predict_proba(X), axis=1)
        else:
            predictions = np.asarray([clf.predict(X) for clf in self.
↪classifiers_]).T

            maj_vote = np.apply_along_axis(lambda x: np.argmax(np.
↪bincount(x,weights=self.weights)),
                                       axis=1,arr=predictions)
        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

    def predict_proba(self, X):
        """ Predict class probabilities for X.

        Parameters
        ----------
        X : {array-like, sparse matrix}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.

        Returns
        ----------
        avg_proba : array-like, shape = [n_samples, n_classes]
            Weighted average probability for each class per sample.

        """
        probas = np.asarray([clf.predict_proba(X) for clf in self.classifiers_])
        avg_proba = np.average(probas, axis=0, weights=self.weights)
        return avg_proba

    def get_params(self, deep=True):
        """ Get classifier parameter names for GridSearch"""
        if not deep:
            return super(MajorityVoteClassifier, self).get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in six.iteritems(self.named_classifiers):
                for key, value in six.iteritems(step.get_params(deep=True)):
                    out['%s__%s' % (name, key)] = value
            return out
```

[ ]:
```
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
```

```
iris = datasets.load_iris()
X, y = iris.data[50:, [1, 2]], iris.target[50:]
le = LabelEncoder()
y = le.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,␣
 ↪random_state=1, stratify=y)
```

[ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

clf1 = LogisticRegression(penalty='l2',C=0.001,random_state=1)

clf2 = DecisionTreeClassifier(max_depth=1,criterion='entropy',random_state=0)

clf3 = KNeighborsClassifier(n_neighbors=1,p=2,metric='minkowski')

pipe1 = Pipeline([['sc', StandardScaler()],['clf', clf1]])
pipe3 = Pipeline([['sc', StandardScaler()],['clf', clf3]])

clf_labels = ['Logistic regression', 'Decision tree', 'KNN']

print('10-fold cross validation:\n')
for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
    scores =␣
 ↪cross_val_score(estimator=clf,X=X_train,y=y_train,cv=10,scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
```

[ ]:
```
mv_clf = MajorityVoteClassifier(classifiers=[pipe1, clf2, pipe3])

clf_labels += ['Majority voting']
all_clf = [pipe1, clf2, pipe3, mv_clf]

for clf, label in zip(all_clf, clf_labels):
    scores =␣
 ↪cross_val_score(estimator=clf,X=X_train,y=y_train,cv=10,scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
```

2.1.1 Curvas ROC

[ ]:
```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

colors = ['black', 'orange', 'blue', 'green']
linestyles = [':', '--', '-.', '-']
for clf, label, clr, ls in zip(all_clf,clf_labels, colors, linestyles):
    # assuming the label of the positive class is 1
    y_pred = clf.fit(X_train,y_train).predict_proba(X_test)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_true=y_test,y_score=y_pred)
    roc_auc = auc(x=fpr, y=tpr)
    plt.plot(fpr, tpr,color=clr,linestyle=ls,label='%s (auc = %0.2f)' % (label,␣
 ↪roc_auc))

plt.legend(loc='lower right')
plt.plot([0, 1], [0, 1],linestyle='--',color='gray',linewidth=2)

plt.xlim([-0.1, 1.1])
plt.ylim([-0.1, 1.1])
plt.grid(alpha=0.5)
plt.xlabel('False positive rate (FPR)')
plt.ylabel('True positive rate (TPR)')
#plt.savefig('images/07_04', dpi=300)
plt.show()
```

[ ]:
```
from itertools import product

sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)

all_clf = [pipe1, clf2, pipe3, mv_clf]

x_min = X_train_std[:, 0].min() - 1
x_max = X_train_std[:, 0].max() + 1
y_min = X_train_std[:, 1].min() - 1
y_max = X_train_std[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))

f, axarr = plt.subplots(nrows=2,ncols=2,sharex='col',sharey='row',figsize=(7,␣
 ↪5))

for idx, clf, tt in zip(product([0, 1], [0, 1]),all_clf, clf_labels):
    clf.fit(X_train_std, y_train)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
```

```
    Z = Z.reshape(xx.shape)

    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)

    axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
                                  X_train_std[y_train==0, 1],
                                  c='blue',
                                  marker='^',
                                  s=50)

    axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
                                  X_train_std[y_train==1, 1],
                                  c='green',
                                  marker='o',
                                  s=50)

    axarr[idx[0], idx[1]].set_title(tt)

plt.text(-3.5, -5.,s='Sepal width [standardized]',ha='center', va='center',␣
 ↪fontsize=12)
plt.text(-12.5, 4.5,s='Petal length [standardized]',ha='center',␣
 ↪va='center',fontsize=12, rotation=90)
#plt.savefig('images/07_05', dpi=300)
plt.show()
```

[ ]:
```
mv_clf.get_params()
```

[ ]:
```
from sklearn.model_selection import GridSearchCV

params = {'decisiontreeclassifier__max_depth':[1, 2],'pipeline-1__clf__C':[0.
 ↪001, 0.1, 100.0]}

grid = GridSearchCV(estimator=mv_clf,param_grid=params,cv=10,scoring='roc_auc')
grid.fit(X_train, y_train)

for r, _ in enumerate(grid.cv_results_['mean_test_score']):
    print("%0.3f +/- %0.2f %r"
          % (grid.cv_results_['mean_test_score'][r],
             grid.cv_results_['std_test_score'][r] / 2.0,
             grid.cv_results_['params'][r]))
```

[ ]:
```
print('Best parameters: %s' % grid.best_params_)
print('Accuracy: %.2f' % grid.best_score_)
```

[ ]:
```
grid.best_estimator_.classifiers
```

[ ]:
```
mv_clf = grid.best_estimator_
```

```
[ ]: mv_clf.set_params(**grid.best_estimator_.get_params())
```

3 Adaptative Boosting (AdaBoost, 1990)

También está formado por clasificadores bases (simples) a los que se les denomina aprendices débiles.

El concepto clave detrás de Boosting (refuerzo) es centrarse en las muestras de entrenamiento que son difíciles de clasificar. Para esto se deja que los aprendices débiles aprendan posteriormente a partir de las muestras de entrenamiento mal clasificadas, para mejorar el rendimiento en conjunto.

El Boosting utiliza subconjuntos aleatorios de muestras de entrenamiento, extraídas (sin reemplazo) desde el conjunto de datos de entrenamiento, de la siguiente manera:

1. Extraer un subconjunto aleatorio de muestras de entrenamiento d_1 sin reemplazo desde el conjunto de entrenamiento D para entrenar un aprendiz débil C_1.
2. Extraer un segundo subconjunto de entrenamiento aleatorio d_2 sin reemplazo desde el conjunto de entrenamiento y añadir el 50% de las muestras que fueron clasificadas erróneamente anteriormente para entrenar a un aprendiz débil C_2.
3. Encontrar las muestras de entrenamiento d_3 en el conjunto de entrenamiento D, en las que C_1 y C_2 no están de acuerdo, para entrenar a un tercer aprendiz débil C_3.
4. Combine los aprendices débiles C_1, C_2 y C_3 mediante una votación por mayoría.

De esta forma, el Boosting puede conducir a una disminución del sesgo, así como también de la varianza.

En la práctica, los algoritmos de Boosting como AdaBoost también son conocidos por su alta varianza, es decir, la tendencia a sobreajustar los datos de entrenamiento.

AdaBoost utiliza el conjunto de entrenamiento completo para entrenar a los aprendices débiles, donde las muestras de entrenamiento son reponderadas en cada iteración para construir un clasificador fuerte que aprende de los errores de los aprendices débiles anteriores en el conjunto.

3.1 Algoritmo AdaBoost

1. Establecer el vector de pesos w con valores uniformes, donde

$$\sum_j w_j = 1$$

2. Para j en m rondas de refuerzo, hacer lo siguiente:
   a. Entrenar un aprendiz débil ponderado: $C_j = train(X, y, w)$.
   b. Predecir las etiquetas de clase: $\hat{y} = predict(C_j, X)$.
   c. Calcular la tasa de error ponderada: $\varepsilon = w \cdot (\hat{y} \neq y)$.
   d. Calcular el coeficiente: $\alpha_j = 0.5log\left(\frac{1-\varepsilon}{\varepsilon}\right)$.
   e. Actualizar las ponderaciones: $w := w \times exp(-\alpha_j \times \hat{y} \times x)$.

f. Normalizar los pesos para que sumen 1: $w = \frac{w}{\sum_j w_j}$.

3. Calcula la predicción final: $\hat{y} = \left(\sum_{j=1}^{m} \left(\alpha_j \times predict(C_j, m)\right) > 0\right)$

[ ]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(penalty='l2',C=0.001,random_state=1)

ada = AdaBoostClassifier(base_estimator=lr,n_estimators=200,learning_rate=0.
↪1,random_state=1)
```

[ ]:
```
ada = ada.fit(X_train, y_train)
y_train_pred = ada.predict(X_train)
y_test_pred = ada.predict(X_test)

ada_train = accuracy_score(y_train, y_train_pred)
ada_test = accuracy_score(y_test, y_test_pred)
print('AdaBoost train/test accuracies %.3f/%.3f'
      % (ada_train, ada_test))

lr = lr.fit(X_train, y_train)
y_train_pred = lr.predict(X_train)
y_test_pred = lr.predict(X_test)

lr_train = accuracy_score(y_train, y_train_pred)
lr_test = accuracy_score(y_test, y_test_pred)
print('LR train/test accuracies %.3f/%.3f'
      % (lr_train, lr_test))
```

[ ]:
```
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),np.arange(y_min, y_max, 0.1))

f, axarr = plt.subplots(1, 2, sharex='col', sharey='row', figsize=(8, 3))

for idx, clf, tt in zip([0, 1],[lr, ada],['LR', 'AdaBoost']):
    clf.fit(X_train, y_train)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    axarr[idx].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx].scatter(X_train[y_train == 0, 0],X_train[y_train == 0,␣
↪1],c='blue', marker='^')
```

```
    axarr[idx].scatter(X_train[y_train == 1, 0],X_train[y_train == 1,␣
 ↪1],c='green', marker='o')
    axarr[idx].set_title(tt)

axarr[0].set_ylabel('Largo Pétalo', fontsize=12)

plt.tight_layout()
#plt.savefig('images/07_11.png', dpi=300, bbox_inches='tight')
plt.show()
```

TAREA FINAL

1) Aplicar AdaBoost con SVM.

2) Aplicar Adaboost con Árbol de Decisión.

[ ]:
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC
```

[ ]:
```
from sklearn.ensemble import AdaBoostClassifier
```