Clase6 IMA539

Alejandro Ferreira Vergara

April 3, 2023

1 Algunos algoritmos de Scikit-Learn

1.1 ¿Qué aprenderemos hoy?

- **SVM kernel No Lineal**
- **Random Forest**
- **K-Vecinos Más Cercanos (KNN)**
- **Implementación con Scikit-Learn**
- **Documentación:** https://scikit-learn.org/stable/

2 SVM Kernel No Lineal

- **Para abordar problemas de clasificación no lineal**

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(1)
X_xor = np.random.randn(200, 2)
y_xor = np.logical_xor(X_xor[:, 0] > 0,X_xor[:, 1] > 0)
y_xor = np.where(y_xor, 1, -1)

plt.scatter(X_xor[y_xor == 1, 0],X_xor[y_xor == 1,␣
 ↪1],c='b',marker='x',label='1')
plt.scatter(X_xor[y_xor == -1, 0],X_xor[y_xor == -1,␣
 ↪1],c='r',marker='s',label='-1')

plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.legend(loc='best')
plt.tight_layout()
#plt.savefig('images/03_12.png', dpi=300)
plt.show()
```

La idea básica detrás de los métodos de kernel (para tratar estos datos linealmente no separables) es crear combinaciones no lineales de las características originales para proyectarlas en un espacio

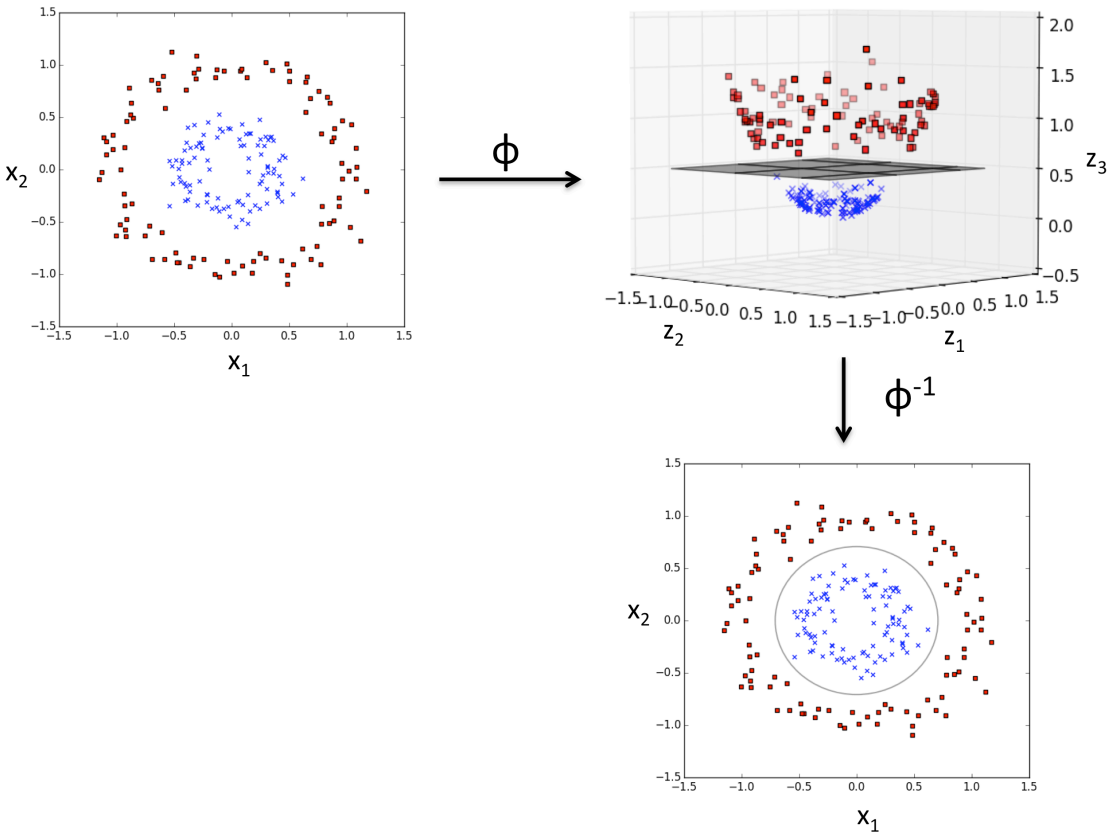
de mayor dimensión a través de una función de mapeo ϕ.

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

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

Image(filename=r'clase6/6_1.png', width=600)
```

[1]:



Para resolver un problema no lineal usando SVM, tenemos que transformar los datos de entrenamiento en un espacio de características de mayor dimensión a través de una función de mapeo ϕ. Luego, podemos encontrar un modelo SVM lineal para clasificar los datos en este nuevo espacio de características. Así, podemos usar la misma función de mapeo ϕ para transformar datos nuevos (no vistos por el modelo) y clasificarlos usando el modelo SVM lineal.

- **Computacionalmente muy costoso.**

Se utiliza el Truco del Kernel, que consiste en reemplazar el producto escalar (en el Dual) $x^{(i)T}x^{(j)}$ **por** $\phi(x^{(i)})^T\phi(x^{(j)})$

- **Definamos la Función Kernel:**

  $K(x^{(i)T}, x^{(j)}) = \phi(x^{(i)})^T\phi(x^{(j)})$

- **Función de Base Radial (RBF o Kernel Gaussiano):**

  $K(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\left\|x^{(i)} - x^{(j)}\right\|^2}{2\sigma^2}\right)$

  $K(x^{(i)}, x^{(j)}) = \exp\left(-\gamma \left\|x^{(i)} - x^{(j)}\right\|^2\right) \in [0, 1]$

  $\gamma = \frac{1}{2\sigma^2}$ es parámetro libre que debe optimizarse.

[ ]:
```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],y=X[y == cl, 1],alpha=0.8,c=colors[idx],
                    marker=markers[idx],label=cl,edgecolor='black')

    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0],X_test[:, 1],c='ghostwhite',edgecolor='black',
                    alpha=1.0,linewidth=1,marker='o',s=100,label='test set')
```

[ ]:
```
from sklearn.svm import SVC

svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
svm.fit(X_xor, y_xor)
plot_decision_regions(X_xor, y_xor,classifier=svm)

plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_14.png', dpi=300)
plt.show()
```

2.1 SVM con datos reales

[ ]:
```
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=1, stratify=y)

print('Cantidad de etiquetas en y:', np.bincount(y))
print('Cantidad de etiquetas en y_train:', np.bincount(y_train))
print('Cantidad de etiquetas en y_test:', np.bincount(y_test))
```

[ ]:
```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
svm.fit(X_train_std, y_train)

X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))

plot_decision_regions(X_combined_std, y_combined,classifier=svm,␣
 ↪test_idx=range(105, 150))
plt.xlabel('largo pétalo [estandarizado]')
plt.ylabel('ancho pétalo [estandarizado]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_15.png', dpi=300)
plt.show()
```

[ ]:
```
svm.predict(X_test_std)
```

[ ]:
```
svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
svm.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std, y_combined,classifier=svm,␣
 ↪test_idx=range(105, 150))
plt.xlabel('largo pétalo [estandarizado]')
```

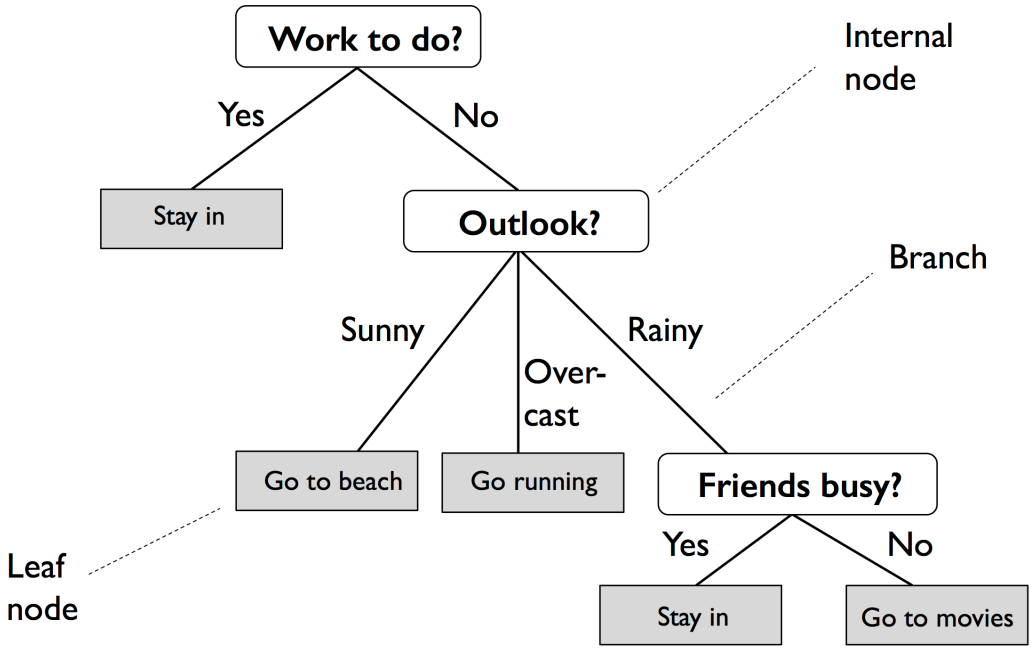
```
plt.ylabel('ancho pétalo [estandarizado]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_16.png', dpi=300)
plt.show()
```

3 Random Forest

3.1 Árboles de Decisión

[2]:
```
Image(filename=r'clase6/6_2.png', width=600)
```

[2]:



Según las características del conjunto de entrenamiento, el modelo de árbol de decisión aprende una serie de preguntas para inferir las etiquetas de clase de las muestras (datos continuos y discretos).

Usando el algoritmo de decisión, **comenzamos en la raíz del árbol y dividimos los datos en la cantidad de ramas que dé como resultado la mayor Ganancia de Información (IG)**.

En un proceso iterativo, podemos repetir este procedimiento de división en cada nodo hijo "hasta que las hojas sean puras". **Esto significa que todas las muestras de cada nodo pertenecen a la misma clase**. En la práctica, esto puede resultar en un árbol muy profundo con muchos nodos, lo que puede conducir fácilmente a un sobreajuste. Por lo tanto, normalmente queremos "podar el árbol" estableciendo un límite máximo para la profundidad del árbol.

Aquí, nuestra **función objetivo es maximizar la Ganancia de Información en cada división**,

que definimos de la siguiente manera:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{n} \frac{N_j}{N_p} I(D_j)$$

f : característica para realizar la división

D_p : conjunto de datos del nodo padre

D_j conjunto de datos del j-ésimo hijo

I : medida de impureza

N_p : número total de muestras en el nodo padre

N_j : número de muestras en el j-ésimo nodo hijo.

Por simplicidad y para reducir el espacio de búsqueda combinatoria, **la mayoría de los algoritmos (incluido el de Scikit-Learn) implementan árboles de decisión binarios**. Esto significa que cada nodo principal se divide en dos nodos secundarios: D_{left} y D_{right}.

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

- **Entropía (para clases no vacías)** $(p(i|t) \neq 0)$**:**

  $I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$

  $p(i|t)$: proporción de las muestras que pertenecen a la clase i para un nodo t en particular.

- **Medida de Impureza de Gini:** Puede entenderse como un criterio para minimizar la probabilidad de clasificación errónea.

  $I_G(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2$

- **Error de clasificación:**

  $I_E = 1 - \max\{p(i|t)\}$

[ ]:
```
def gini(p):
    return p * (1 - p) + (1 - p) * (1 - (1 - p))

def entropy(p):
    return - p * np.log2(p) - (1 - p) * np.log2((1 - p))

def error(p):
    return 1 - np.max([p, 1 - p])

x = np.arange(0.0, 1.0, 0.01)

ent = [entropy(p) if p != 0 else None for p in x]
sc_ent = [e * 0.5 if e else None for e in ent]
err = [error(i) for i in x]
```

```
fig = plt.figure()
ax = plt.subplot(111)
for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
                          ['Entropy','Entropy (scaled)','Gini␣
 ↪Impurity','Misclassification Error'],
                          ['-', '-', '--', '-.'],
                          ['black', 'blue', 'red', 'green', 'cyan']):
    line = ax.plot(x, i, label=lab, linestyle=ls, lw=2, color=c)

ax.legend(loc='upper center',bbox_to_anchor=(0.5, 1.
 ↪15),ncol=5,fancybox=True,shadow=False)

ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
plt.ylim([0, 1.1])
plt.xlabel('p(i=1)')
plt.ylabel('Impurity Index')
#plt.savefig('images/03_19.png', dpi=300, bbox_inches='tight')
plt.show()
```

[ ]:
```
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier(criterion='gini', max_depth=4, random_state=1)
tree.fit(X_train, y_train)

X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))

plot_decision_regions(X_combined, y_combined,classifier=tree,␣
 ↪test_idx=range(105, 150))

plt.xlabel('largo pétalo [estandarizado]')
plt.ylabel('ancho pétalo [estandarizado]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_20.png', dpi=300)
plt.show()
```

3.2 Combinando múltiples árboles de decisión con Random Forest

La idea detrás de Random Forest es promediar múltiples árboles de decisión (profundos) que individualmente sufren de alta varianza, para construir un modelo más robusto que tenga un mejor rendimiento de generalización y sea menos susceptible al sobreajuste. **El algoritmo de Random Forest lo podemos resumir en cuatro pasos:**

1. Elejir aleatoriamente n muestras del conjunto de entrenamiento (con reemplazo).

2. Hacer crecer un árbol de decisiones a partir de una muestra de partida. En cada nodo:
    - Seleccionar d características al azar sin reemplazo.
    - Dividir el nodo utilizando la función que proporcione la mejor división de acuerdo con la función objetivo.
3. Repitir los pasos 1-2 k veces.
4. Agregar la predicción por cada árbol para asignar la etiqueta de clase por mayoría de votos.

Es un algoritmo no tan sensible a los Hiperparámetros. Por lo general, no necesitamos "*podar el bosque aleatorio*" ya que el modelo en conjunto es bastante robusto al ruido de los árboles de decisión individuales. El único parámetro que realmente debemos preocuparnos (en la práctica) es el número de árboles k (paso 3) que elegimos.

NOTA: En la mayoría de las implementaciones, incluida la implementación de *RandomForestClassifier* en Scikit-Learn, el tamaño de la muestra de partida se elige para que sea igual al número de muestras en el conjunto de entrenamiento original, lo que generalmente proporciona una buena compensación de sesgo-varianza. Para el número de características d en cada división, queremos elegir un valor que sea menor que el número total de características en el conjunto de entrenamiento. Un valor predeterminado razonable que se usa en Scikit-Learn y otras implementaciones es $d = \sqrt{m}$, donde m es el número de características en el conjunto de entrenamiento.

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

forest = RandomForestClassifier(criterion='gini', n_estimators=25,␣
 ↪random_state=1, n_jobs=-1)
forest.fit(X_train, y_train)

plot_decision_regions(X_combined, y_combined,classifier=forest,␣
 ↪test_idx=range(105, 150))

plt.xlabel('largo pétalo [estandarizado]')
plt.ylabel('ancho pétalo [estandarizado]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_22.png', dpi=300)
plt.show()
```

4 K-Vecinos Más Cercanos (KNN)

- **Ejemplo de aprendíz peresoso. Memoriza el conjunto de datos de entrenamiento.**
- **Modelo No Paramétrico subclasificado como de Aprendizaje Basado en instancias.**

Algoritmo:

1. Elejir el número k y una métrica de distancia.
2. Encontrar los k vecinos más cercanos de la muestra que se quiere clasificar.
3. Asignar la etiqueta de la clase por mayoría de votos (clase más representada).

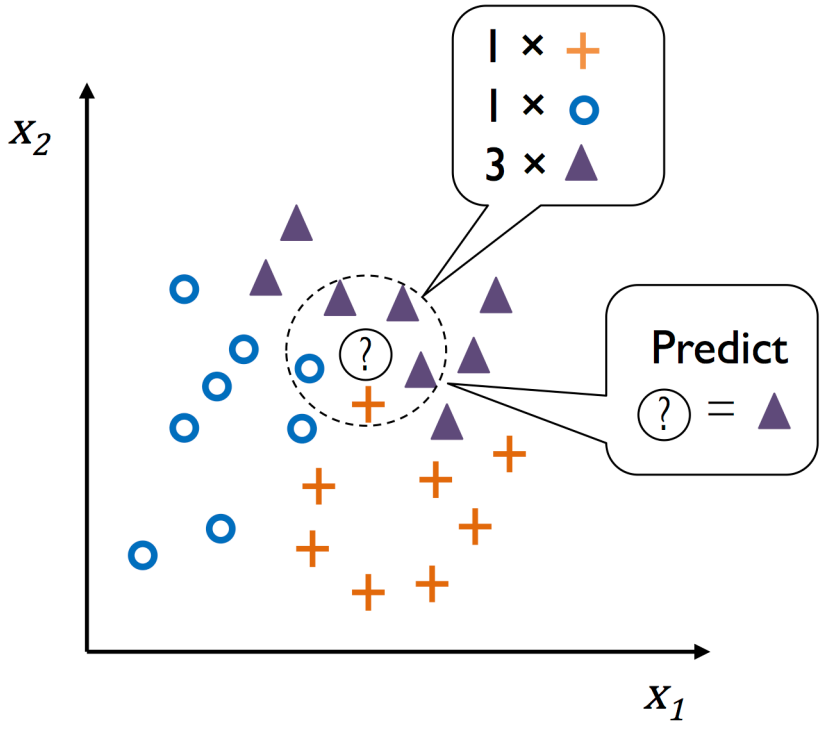
Según la métrica de distancia elegida, el algoritmo **KNN** encuentra las k muestras, en el conjunto de datos de entrenamiento, que están más cerca (más similares) al punto que queremos clasificar.

Ventaja: El clasificador se adapta inmediatamente a medida que recopilamos nuevos datos de entrenamiento.

Desventaja: La complejidad computacional para clasificar nuevas muestras crece linealmente con la cantidad de muestras en el conjunto de datos de entrenamiento en el peor de los casos, a menos que el conjunto de datos tenga muy pocas dimensiones (características) y el algoritmo se haya implementado utilizando datos eficientes.

[3]:
```
Image(filename=r'clase6/6_3.png', width=450)
```

[3]:



[ ]:
```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski', n_jobs=-1)
knn.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std, y_combined,classifier=knn,␣
 ↪test_idx=range(105, 150))

plt.xlabel('largo pétalo [estandarizado]')
plt.ylabel('ancho pétalo [estandarizado]')
plt.legend(loc='upper left')
```

```
plt.tight_layout()
#plt.savefig('images/03_24.png', dpi=300)
plt.show()
```

[ ]: