Implementacion de clasificadores de Scikit-learn en la base de datos Glass.

Echeverry Luis (2420161006)
2420161006@estudiantesunibague.edu.co
Inteligencia Artificial
Programa de Ingeniería Electrónica
Facultad de Ingeniería
Universidad de Ibagué
Ibagué - Tolima
Semestre 2020A
Presentado al ingeniero Jose Armando Fernández Gallego

1. Resumen

Para esta entrega se tiene como objetivo general, continuar con la implementación de los algoritmos de clasificación que se vieron durante el curso en la base de datos seleccionada. Como objetivo específico se tiene la separación de al menos 3 clases presentes en la base de datos, con dos de sus características, haciendo uso de las diferentes herramientas que nos ofrece la librería Scikit-learn de Python. El desempeño de los diferentes algoritmos se evaluará, teniendo en cuenta el porcentaje de clasificación y la gráfica de las regiones de separación, lo que permitirá de esta manera sacar algunas conclusiones importantes acerca de los diferentes algoritmos presentes en este informe. Los clasificadores que se desean implementar son Perceptron, Perceptron + regresión logística, SVM + linear, SVM + RBF, DT, RF. Los cuales están contenidos en la librería Scikit-learn mencionada previamente.

2. Desarrollo

Como se mencionó previamente, es indispensable describir la base de datos seleccionada. Glass Identification Data Set es una recopilación de las características de fragmentos de vidrios obtenidos en escenas de crímenes, realizada por el departamento de ciencias forenses de los Estados Unidos. Está compuesta por un número total de 214 muestras de 6 tipos diferentes de vidrios. Las cuales, a su vez, tienen los valores correspondientes del contenido de óxido de Sodio (Na), Magnesio (Mg), Aluminio (Al), Bario (Ba), Silicón (Si), Potasio (K), Calcio (Ca) y Hierro (Fe). Además del índice de refracción y un número único para cada muestra tomada, parámetros que fueron evaluados al momento de realizar las pruebas necesarias para crear cada una de las muestras.

La razón de la creación de esta base de datos, fue el diseño de un algoritmo que permita clasificar de forma rápida y segura el tipo de vidrio encontrado en una escena del crimen o de un accidente, para que este pueda llegar a servir como evidencia, de ser necesario.

Para este caso en específico se decidió hacer uso del Aluminio (Al) y Sodio (Na) ya que como se puede observar en figura 1, son las dos clases que más se ajustan a una separación lineal entre las tres clases de vidrios a seleccionar.

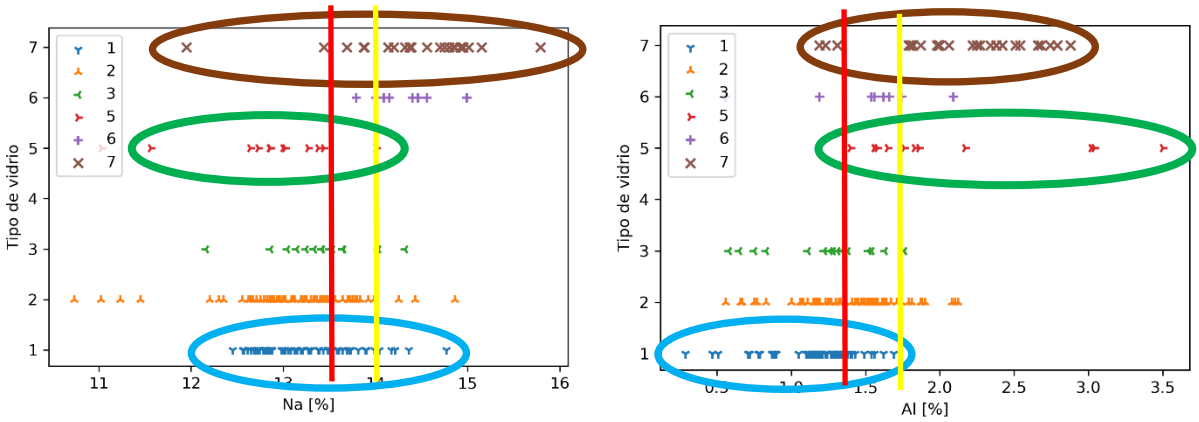


Figura 1. Grafica de Na y Al en cada uno de los 6 tipos de vidrios.

Como segunda, tenemos que los algoritmos que se desean aplicar son el perceptrón, el perceptrón más la regresión logística, máquinas de soporte vectorial (SVM) de forma lineal, máquinas de soporte vectorial (SVM) con kernel de función de base radial, los árboles de decisión y los random forests. Cada uno de estos se encuentra dentro de la librería Scikit-learn de Python, facilitando asi su implementación.

3. Diagrama de flujo

A continuación, se presenta el diagrama de flujo que corresponde a la implementación de cada uno de los clasificadores mencionados anteriormente, especificando sus correspondientes salidas y entradas.

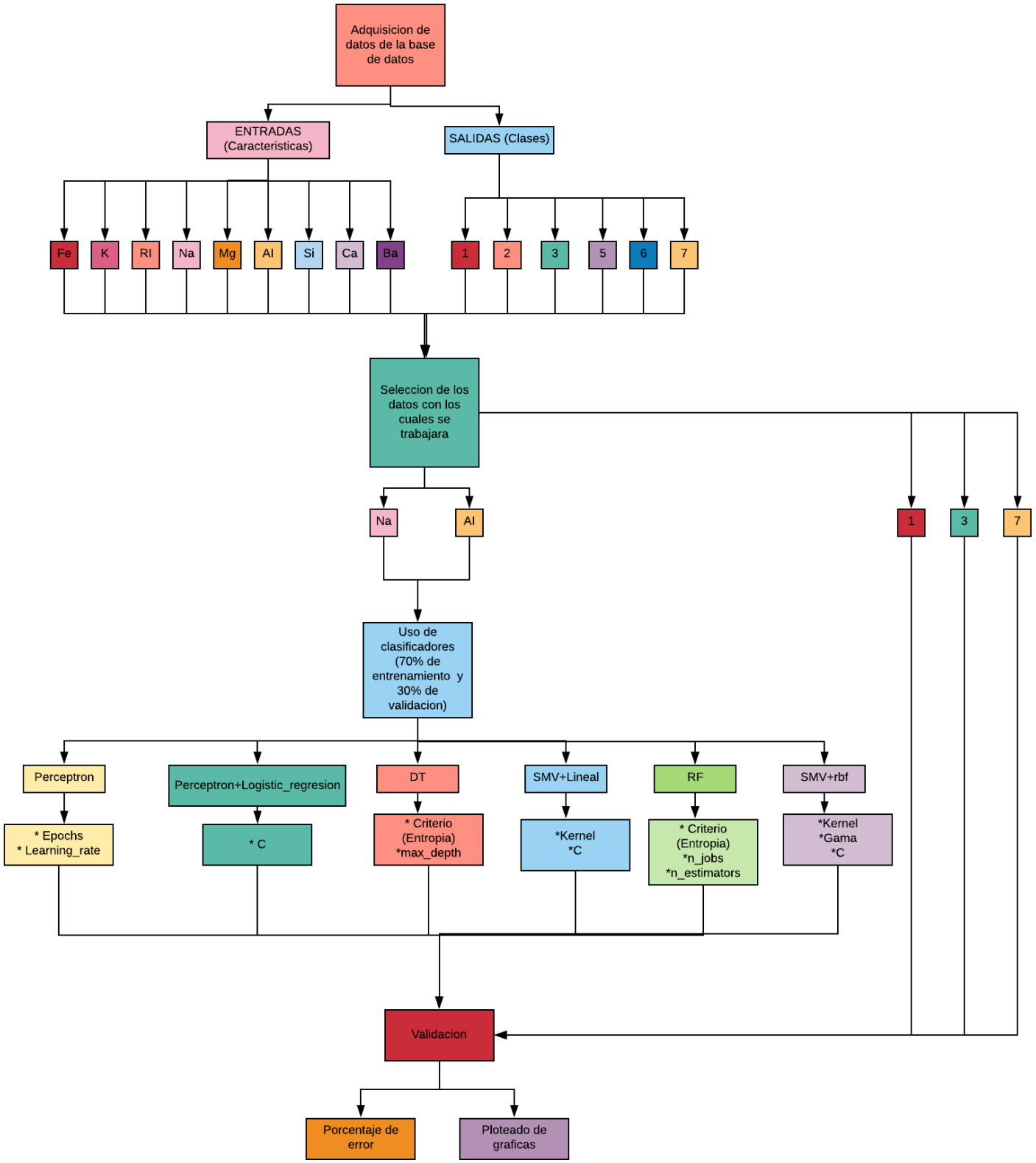


Figura 2. Diagrama de flujo del código a implementar.

4. Resultados

Perceptrón.

Se decidió iniciar por el algoritmo del perceptrón, ya que este fue analizado con mayor profundidad durante el segundo corte de la materia. De forma general, este algoritmo toma los valores de entrada y los multiplica por unos pesos, los cuales se actualizan de forma iterativa en cada época. Al evaluar cada uno de estos valores, se realiza la sumatoria de todos ellos y se hacen pasar a través de una función de activación que, para el caso del perceptrón normal, es una función de escalón unitario. Con el resultado obtenido de evalúa el error y dependiendo de este se actualiza nuevamente cada uno de los pesos.

Para el caso particular del perceptrón en la librería Scikit-learn de Python, hace falta escalar los datos de entrada de la misma manera que se hizo en la entrega anterior. Ya que como se afirma en el libro guía, la estandarización es indispensable para el buen desempeño de algunos algoritmos de clasificación. Para esto, la librería nos presenta las herramientas necesarias para realizar esta tarea, ofreciéndonos la posibilidad de remplazar varias líneas de código por las siguientes:

```
111    # Estandarizacion
112    sc = StandardScaler()
113    sc.fit(X_train)
114    X_train_std = sc.transform(X_train)
115    X_test_std = sc.transform(X_test)
```

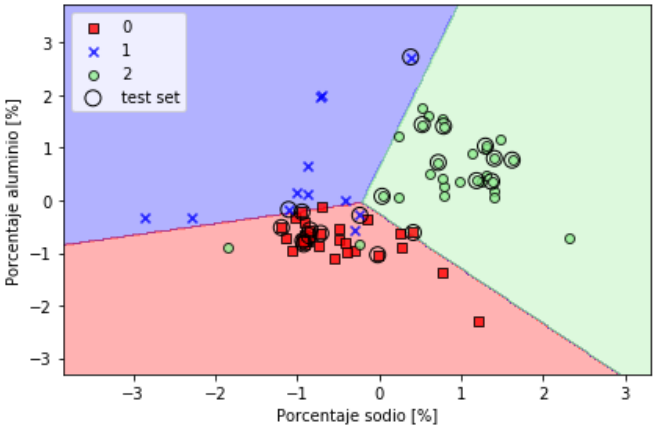
Figura 3. Estandarización en Python.

Con esto ya se hace posible la utilización del perceptrón de la librería Scikit-learn de Python como se muestra en las siguientes líneas de código:

```
121    #PERCEPTRON
122    ppn = Perceptron(max_iter=40, eta0=0.1, random_state=1)
123    ppn.fit(X_train_std, y_train)
124
125        #Errores
126    y_pred = ppn.predict(X_test_std)
127    print('Misclassified samples of PPN: %d' % (y_test != y_pred).sum())
128    print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
129    print('Accuracy: %.2f' % ppn.score(X_test_std, y_test))
130
131        #regiones de decisión
132    plot_decision_regions(X=X_combined_std, y=y_combined,
133                          classifier=ppn, test_idx=range(48, 69))
134    plt.xlabel('Porcentaje sodio [%]')
135    plt.ylabel('Porcentaje aluminio [%]')
136    plt.legend(loc='upper left')
137
138    plt.tight_layout()
139    #plt.savefig('images/03_01.png', dpi=300)
140    plt.show()
```

Figura 4. Implementación del perceptrón en Python.

Donde al igual que en la implementación que se realizó en entregas anteriores, los parámetros de entrada son el número de iteraciones y la taza de aprendizaje. Al evaluar el algoritmo de esta manera obtuvimos los siguientes resultados:




```
Misclassified samples of PPN: 2
Accuracy: 0.90
Accuracy: 0.90
```

Figura 5. Resultados de implementación del perceptrón en Python.

Con lo anterior podemos analizar que, de forma general, la clasificación de los datos ha sido realizada de una manera correcta. Las regiones fueron separadas de tal manera que el mayor numero de datos estuviera dentro de ellas. Pero aquí es donde se pueden observar algunas de las limitantes de este algoritmo, ya que la separación que realiza el perceptrón es de forma lineal, lo que provoca que, si las tres clases no se pueden separar perfectamente con una línea, el error nunca convergerá a cero y los pesos se actualizaran indefinidamente. A pesar de esto, se puede observar que el porcentaje de error fue muy pequeño, ya que de 69 muestras que fueron tomadas, solo 2 fueron clasificadas de forma errónea, dando un acierto de más del 97%.

Perceptrón más la regresión logística.

Para continuar, se decidió seguir con el mismo patrón que lleva el libro, y en este caso se continuo por la modificación del algoritmo del perceptrón, cambiando la función de activación del escalón unitario por la sigmoide. Esto permite que la regresión logística sea capaz de indicar la probabilidad de que un dato pertenezca a una clase específica, a diferencia del perceptrón, el cual solo actualizaba los pesos y basado en esto hacia su predicción.

Ya que este algoritmo basa sus resultados en un modelo probabilístico, podemos inferir de antemano que va a tener un mejor desempeño en la clasificación de los parámetros de nuestra base de datos. Para la implementación, vamos a hacer uso nuevamente de la librería Scikit-learn de Python como se muestra a continuación:

```
#PERCEPTRON + LOGISTIC REGRESSION
lr = LogisticRegression(C=10.0, random_state=1)
lr.fit(X_train_std, y_train)
    #Errores
y_pred = lr.predict(X_test_std)
print('Misclassified samples of PPN+LR: %d' % (y_test != y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
print('Accuracy: %.2f' % lr.score(X_test_std, y_test))
    #regiones de decisión
plot_decision_regions(X_combined_std, y_combined,
                      classifier=lr, test_idx=range(48, 69))
plt.xlabel('Porcentaje sodio [%]')
plt.ylabel('Porcentaje aluminio [%]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_06.png', dpi=300)
plt.show()
```

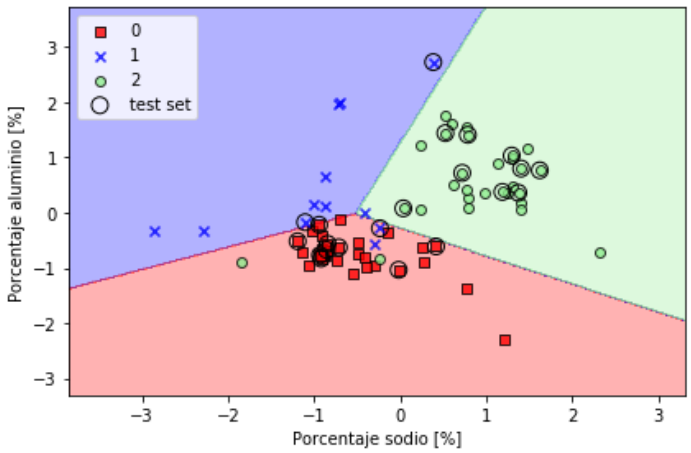
Figura 6. Implementación del perceptrón + LR en Python.

Donde de manera intuitiva podemos observar que el único parámetro que se debe definir para la implementación del este algoritmo es el parámetro C. Este es el responsable de evitar un sobre ajuste en de la red, ya que es el inverso del parámetro de regularización λ, tal y como se muestra en la siguiente ecuación:

$$C = \frac{1}{\lambda}$$

Ecuación 1. Relación ente C y el parámetro de regularización.

Con esto claro, a continuación se presentan los resultados obtenidos:




```
Misclassified samples of PPN+LR: 1
Accuracy: 0.95
Accuracy: 0.95
```

Figura 7. Resultados de implementación del perceptrón + LR en Python.

Observando la gráfica anterior, podemos analizar qué, tal y como se esperaba, el desempeño del algoritmo fue mucho mejor que el del perceptrón, ya que el número de parámetros clasificados erróneamente fue menor, haciendo que el porcentaje de acierto subiera del 97% al 98.5%. a pesar de que se pudo ver una gran mejora en el algoritmo, podemos analizar que aun persiste un error en la

clasificación, esto se debe a que la regresión logística sigue siendo un método de separación de clases lineales.

Máquinas de soporte vectorial.

El siguiente algoritmo que se decidió analizar, es el de las máquinas de soporte vectorial o SVM por sus siglas en inglés. Este tiene como objetivo minimizar el margen que existe entre el hiperplano positivo y negativo. De esta manera de podría decir que las maquinas de soporte vectorial son una modificación adicional que se le realiza al algoritmo del perceptrón, para que este, además de determinar una línea o separación entre los conjuntos de datos, pueda maximizar ese margen hasta que este se encuentre equidistante a los vectores de soporte. Este concepto se ve más claramente en la siguiente gráfica:

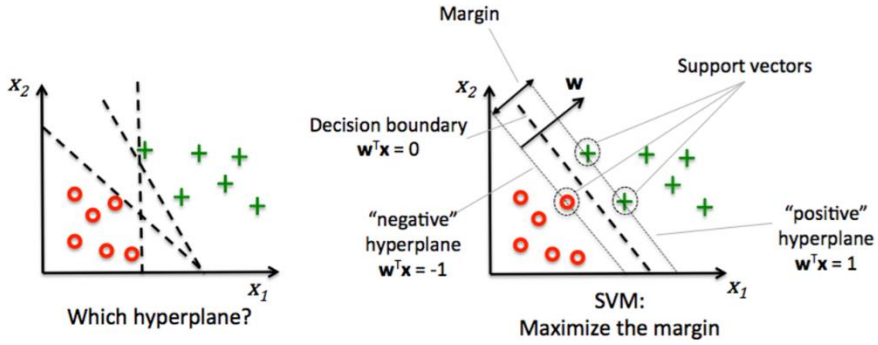


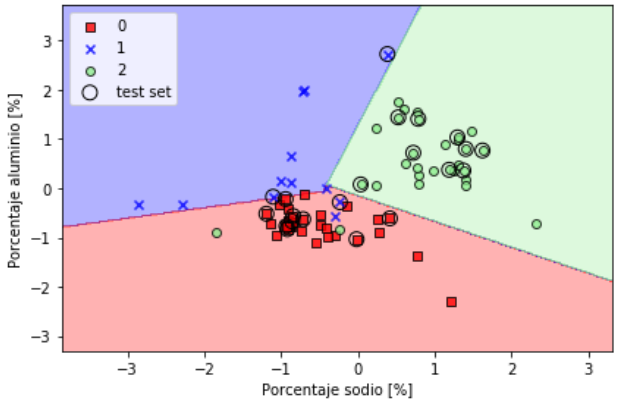
Figura 8. Principio lógico de SVM.

Para la implementación en Python, los parámetros que se deben definir son el kernel, del cual se hablara con mayor profundidad en el siguiente algoritmo, y C, parámetro el cual se mencionó en la sección anterior. se realiza de la siguiente manera:

```
161  #SUPPORT VECTOR MACHINE
162  svm = SVC(kernel='linear', C=100.0, random_state=1)
163  svm.fit(X_train_std, y_train)
164    #Errores
165  y_pred = svm.predict(X_test_std)
166  print('Misclassified samples of SVM: %d' % (y_test != y_pred).sum())
167  print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
168  print('Accuracy: %.2f' % svm.score(X_test_std, y_test))
169      #regiones de decisión
170  plot_decision_regions(X_combined_std,
171                        y_combined,
172                        classifier=svm,
173                        test_idx=range(48, 69))
174  plt.xlabel('Porcentaje sodio [%]')
175  plt.ylabel('Porcentaje aluminio [%]')
176  plt.legend(loc='upper left')
177  plt.tight_layout()
178  #plt.savefig('images/03_11.png', dpi=300)
179  plt.show()
```

Figura 9. Implementación de SVM en Python.

Con lo anterior podemos inferir que, a pesar de que las maquinas de soporte vectorial son algoritmos un poco mas robustos que el perceptrón o la regresión logística, estos siguen siendo lineales, ya que lo que se busca es separar de forma lineal las tres clases presentes en nuestra base de datos. Es por esta razón que el resultado de la implementación del algoritmo es muy similar al de la regresión logística, tal y como se muestra a continuación:




```
Misclassified samples of SVM: 1
Accuracy: 0.95
Accuracy: 0.95
```

Figura 10. Resultados de implementación de SVM en Python.

De lo anterior se puede inferir que el a regresión logística tiene la ventaja, sobre las maquinas de soporte vectorial, de que es un modelo más simple que puede implementarse más fácilmente, a pesar de que este tuvo un porcentaje de error del 98.5% al igual que el algoritmo anterior.

Máquinas de soporte vectorial con Kernel de base radial.

Para continuar, procedemos a implementar el mismo algoritmo con la diferencia que se le aplicara un tipo diferente de Kernel. La idea básica detrás de los métodos del núcleo para tratar con estos datos inseparables linealmente es crear combinaciones no lineales de las características originales para proyectarlas en un espacio dimensional superior a través de una función de mapeo.

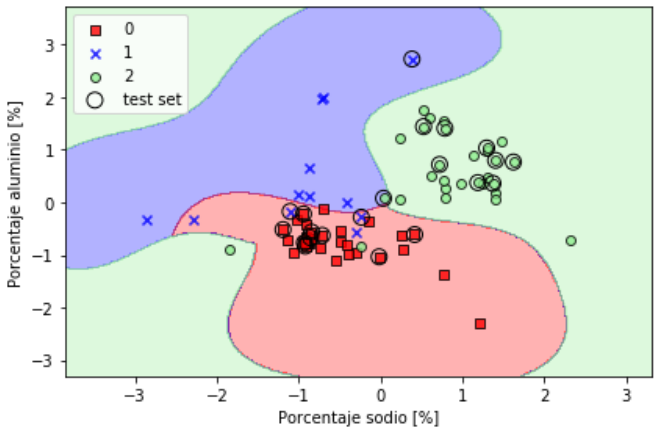
Con esto se logra que la separación de los datos se realice en un tercer plano diferente a los dos presentes por defecto. La implementación de este algoritmo en Python re realiza como se muestra a continuación:

```
#SUPPORT VECTOR MACHINES + RBF
svm = SVC(kernel='rbf', random_state=1, gamma=0.01, C=100.0)
svm.fit(X_train_std, y_train)
  #Errores
y_pred = svm.predict(X_test_std)
print('Misclassified samples of SVM + RBF: %d' % (y_test != y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
print('Accuracy: %.2f' % svm.score(X_test_std, y_test))
    #regiones de decisión

plot_decision_regions(X_combined_std, y_combined,
                      classifier=svm, test_idx=range(48, 69))
plt.xlabel('Porcentaje sodio [%]')
plt.ylabel('Porcentaje aluminio [%]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_16.png', dpi=300)
plt.show()
```

Figura 9. Implementación de SVM + RBF en Python.

Donde el parámetro γ es el parámetro de corte para la esfera gaussiana. Si este se aumenta, la influencia o alcance de las muestras de entrenamiento aumentan también. Una mala elección de este parámetro puede llevar a un sobre entrenamiento de la red y a obtener regiones de separación erróneas. Por esta razón, este parámetro se ajustó en 0.01 obteniendo los siguientes resultados:




```
Misclassified samples of SVM + RBF: 1
Accuracy: 0.95
Accuracy: 0.95
```

Figura 11. Resultados de implementación de SVM +RBF en Python.

Con lo cual podemos observar que las regiones de separación se ajustaron de forma no lineal a los datos, lo que se presume que permitirá una mejor clasificación de los mismo, aunque esto no se vio reflejado en este conjunto de datos, ya que el porcentaje de aciertos se mantuvo en 98.5%.

Arboles de decisión.

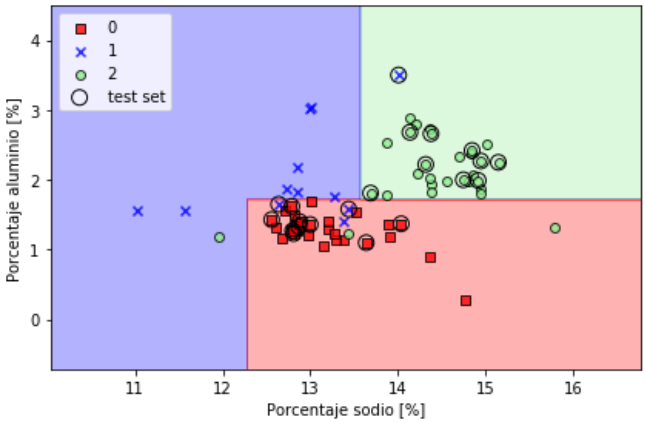
Para el caso de los arboles de decisión, el objetivo es separar en regiones rectangulares cada una de las clases presentes en nuestro conjunto de datos. El parámetro más importante en la implementación de los árboles de decisión es la profundidad. Este parámetro define cuantos nodos va a tener nuestro algoritmo y juega un papel fundamental en el objetivo de evitar un sobre ajuste de nuestra red. La implementación en Python se ve a continuación:

```
#DECISION TREE
tree = DecisionTreeClassifier(criterion='gini',
                              max_depth=2,
                              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))
  #Errores
y_pred = tree.predict(X_test)
print('Misclassified samples of DT: %d' % (y_test != y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
print('Accuracy: %.2f' % tree.score(X_test_std, y_test))
    #regiones de decisión
plot_decision_regions(X_combined, y_combined,
                      classifier=tree, test_idx=range(48, 69))
plt.xlabel('Porcentaje sodio [%]')
plt.ylabel('Porcentaje aluminio [%]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_20.png', dpi=300)
plt.show()
```

Figura 9. Implementación de SVM + RBF en Python.

Los resultados obtenidos se muestran a continuación:




```
Misclassified samples of DT: 2
Accuracy: 0.90
Accuracy: 0.14
```

Figura 9. Implementación de SVM + RBF en Python.

Con los resultados obtenido podemos observar que las regiones de clasificación se realizaron de forma correcta pero el porcentaje de aciertos se redujo del 98.5% al 97%. Esto se debe a que la profundidad de nuestro algoritmo fue de apenas 2 nodos con el fin de evitar un sobre ajuste de nuestro sistema.

Random forests.

En términos generales, este algoritmo permite realizar una clasificación mas precisa gracias a que es sumamente robusto con respecto al ruido. Consiste en extraer pequeños grupos de muestras aleatorias del conjunto de entrenamiento y generar un árbol de decisión para cada uno de ellas, determinado así, la clase predicha gracias a la mayor votación.

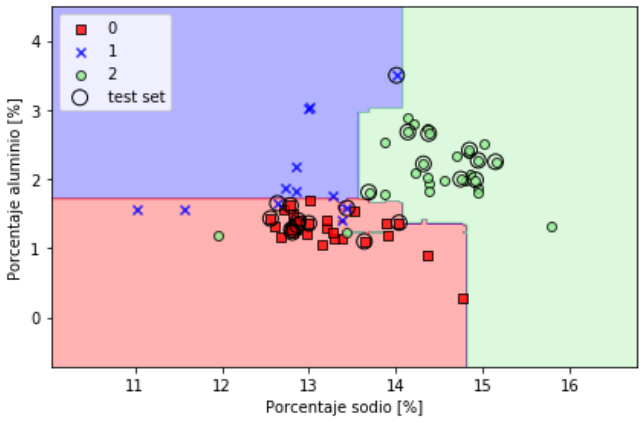
Una gran ventaja de los Random Forest, es que no tenemos que preocuparnos tanto por elegir buenos valores de hiperparámetros. Por lo general, no necesitamos podar el bosque aleatorio ya que el modelo de conjunto es bastante robusto al ruido de los árboles de decisión individuales. El único parámetro que realmente debemos preocuparnos en la práctica es el número de árboles k (paso 3) que elegimos para el bosque aleatorio. Por lo general, cuanto mayor sea el número de árboles, mejor será el rendimiento del clasificador forestal aleatorio a expensas de un mayor costo computacional.

La implementación en Python se ve de la siguiente manera:

```
#RANDOM FORESTS
forest = RandomForestClassifier(criterion='gini',
                                n_estimators=10,
                                random_state=1,
                                n_jobs=2)
forest.fit(X_train, y_train)
  #Errores
y_pred = forest.predict(X_test)
print('Misclassified samples of RF: %d' % (y_test != y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
print('Accuracy: %.2f' % forest.score(X_test_std, y_test))
    #Regiones de decisión
plot_decision_regions(X_combined, y_combined,
                      classifier=forest, test_idx=range(48, 69))
plt.xlabel('Porcentaje sodio [%]')
plt.ylabel('Porcentaje aluminio [%]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_22.png', dpi=300)
plt.show()
```

Figura 9. Implementación de SVM + RBF en Python.

Donde el parámetro *n_estimators* es el que determina el número de árboles que tendrá nuestro algoritmo. En este caso en particular, decidimos utilizar 10 arboles con el fin de optimizar el procesamiento de nuestros datos. Con lo anterior tenemos que los resultados son los siguientes:




```
Misclassified samples of RF: 3
Accuracy: 0.86
Accuracy: 0.43
```

Figura 9. Implementación de SVM + RBF en Python.

Con los datos obtenidos anteriormente, podemos concluir que este clasificador es de gran ayuda, ya que logra separar las clases de nuestra base de datos de forma correcta. Por otro lado también se podría afirmar, que este algoritmo no es muy eficiente para la implementación de esta base de datos específica, ya que a pesar de que se realiza una separación correcta de las clases, el costo computacional es muy elevado y los resultados obtenidos se pueden obtener con otros algoritmos menos demandantes.

5. Conclusiones.

- La librería Scikit-learn de Python es una herramienta muy útil en el desarrollo de algoritmos para el proceso de clasificación y separación de grupos de datos, facilitando el trabajo del programador.
- El peligro de que una red sufra de sobre entrenamiento se encuentra latente, a pesar de que los algoritmos tratados siempre han sido orientados a evitar caer en este problema.
- El algoritmo que mas se ajusta a la base de datos utilizada fue el de los árboles de decisión, ya que se equilibran de forma adecuada el costo computacional con el resultado exitoso de clasificación.