Ejercicios SKLEARN

Dataset

He elegido como dataset “Glass Identification” en el que tendremos como se clasifican ciertos tipos de cristal dependiendo de sus características. Tenemos las siguientes columnas:

- **Id_number**: Nos indica el id de cada fila. Este lo vamos a quitar porque ya el propio pandas nos da este valor
- **RI**: El indice de refraccion
- **Na**: El porcentaje en peso en el óxido correspondiente. En este caso sodio
- **Mg**: El porcentaje en peso en el óxido correspondiente. En este caso magnesio
- **Al**: El porcentaje en peso en el óxido correspondiente. En este caso aluminio
- **Si**: El porcentaje en peso en el óxido correspondiente. En este caso silicio
- **K**: El porcentaje en peso en el óxido correspondiente. En este caso potasio
- **Ca**: El porcentaje en peso en el óxido correspondiente. En este caso calcio
- **Ba**: El porcentaje en peso en el óxido correspondiente. En este caso bario
- **Fe**: El porcentaje en peso en el óxido correspondiente. En este caso hierro
- **Type_of_glass**: Este sera el valor que queremos clasificar y tendremos los siguientes tipos:
    1. building_windows_float_processed
    2. building_windows_non_float_processed
    3. vehicle_windows_float_processed
    4. vehicle_windows_non_float_processed (none in this database)
    5. containers
    6. tableware
    7. headlamps

Lo bueno de este dataset es que los datos no necesitan de ningún tipo de tratamiento al no faltar valores y venir todos en formato numérico. Lo único que hacemos es eliminar la columna Id_number. Lo malo es que es un dataset bastante pequeño, lo cual nos afecta al rendimiento a la hora de clasificar.

Separar Dataset

Tras cargar el dataset, lo primero que tenemos que hacer es separarlo en los conjuntos de entrenamiento y test. Así que separamos los datos(x) de la variable clasificadora(y) y le aplicamos la función “train_test_split” para tener nuestro conjuntos:

```
x = df.iloc[:, :8].values
```

```
y = df.iloc[:, 9].values
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=0)
```

En este caso he optado por un 80% de datos para entrenar y el restante 20% para test. Con random_state me aseguro que los datos sean iguales entre ejecuciones, al funcionar este como seed para el generador de números pseudoaleatorios.

Por ultimo, creamos una variable con la función “StratifiedKFold” con 10-fold:

```
skf = StratifiedKFold(n_splits=10)
```

Esto lo usaremos a la hora de encontrar los mejores hiperparametros para los distintos algoritmos de manera que todos tengan las mismas condiciones.

Naïve Bayes

Como clasificador he usado en concreto el modelo Gaussiano:

```
gnb = GaussianNB()
```

```
gnb = gnb.fit(x_train, y_train)
```

```
y_pred = gnb.predict(x_test)
```

```
y_pred = gnb.fit(x_train, y_train).predict(x_test)
```

Para este no he usado ningún parámetro ya no que lo he visto necesario. Así que creamos una variable con el clasificar, lo ajusto a los datos con “.fit()” y predecimos.

Compruebo los resultados con:

```
print("Cantidad de predicciones mal hechas de %d valores : %d" % (x_test.shape[0], (y_test != y_pred).sum()))
```

```
print("Accuracy "+str((y_test != y_pred).sum()*100/x_test.shape[0])+"%")
```

El primero print me dice cuantos ha fallado y el segundo me dice la precisión. El primero print lo iré usando en el resto de algoritmos para tener una forma clara de ver como de bien predice.

Al ejecutar el código nos dice:

```
Cantidad de predicciones mal hechas de 43 valores : 27
```

```
Accuracy 62.7906976744186%
```

Nearest Neighbors

Para usar este algoritmo de manera efectiva, primero hay que encontrar cual es el mejor numero de vecinos a utilizar como hiperparametro. Lo hago de esta manera:

```
gsVecinos = GridSearchCV(
    KNeighborsClassifier(),
    param_grid={"n_neighbors":range(1,101)},
    scoring='accuracy',
    cv = skf
)
```

Donde la función “GridSearchCV()” buscara de manera automática cuales son los mejores parámetros de entre los que le digamos que busque para el algoritmo que queramos. Esto lo hace usando un sistema de scoring donde puedes elegir muchos tipos, pero para este caso he usado simplemente la precisión. Ademas, usara se le puede decir que tipo de Cross-Validation usar, así que le paso el “StratifiedKFold” con 10-fold que he hecho al principio.

Tras esto, entrenamos el modelo con nuestros datos y nos dirá su precisión, los mejores parámetros y podemos pedirle que nos devuelve el mejor modelo para que podamos empezar a predecir datos:

```
gsVecinos.fit(x_train, y_train)
print("El mejor numero de vecinos es: ",gsVecinos.best_params_['n_neighbors'])
print("Mejor precision: ", gsVecinos.best_score_)
vecinosMejor = gsVecinos.best_estimator_
y_pred = vecinosMejor.predict(x_test)
print("Cantidad de predicciones mal hechas de %d valores : %d" % (x_test.shape[0], (y_test != y_pred).sum()))
```

Si ejecutamos el código vemos lo siguiente:

```
El mejor numero de vecinos es:  1
Mejor precision:  0.742156862745098
Cantidad de predicciones mal hechas de 43 valores : 15
```

Decision Tree

Los arboles de decisión son un tipo de algoritmo en SKLearn que podemos personalizar de bastantes maneras. Así que hago un diccionario con los parámetros que quiero que pruebe:

```
parámetros = {
    "criterion" : ["gini", "entropy", "log_loss"],
    "splitter" : ["best", "random"],
    "max_depth" : [None],
    "min_samples_split": [2],
    "min_samples_leaf": [1],
    "max_features": [10],
    "class_weight": [None, "balanced"],
    "ccp_alpha": [0.0, 0.1, 0.2, 0.5]
}
```

Algunos de ellos solo tienen un valor porque, tras probar varias veces con distintos valores, siempre me daban el mínimo, máximo o default. Así que para ahorrar recursos solo he puesto esos. Ademas, faltan parámetros que he considerado que empeoraban el resultado del clasificador.

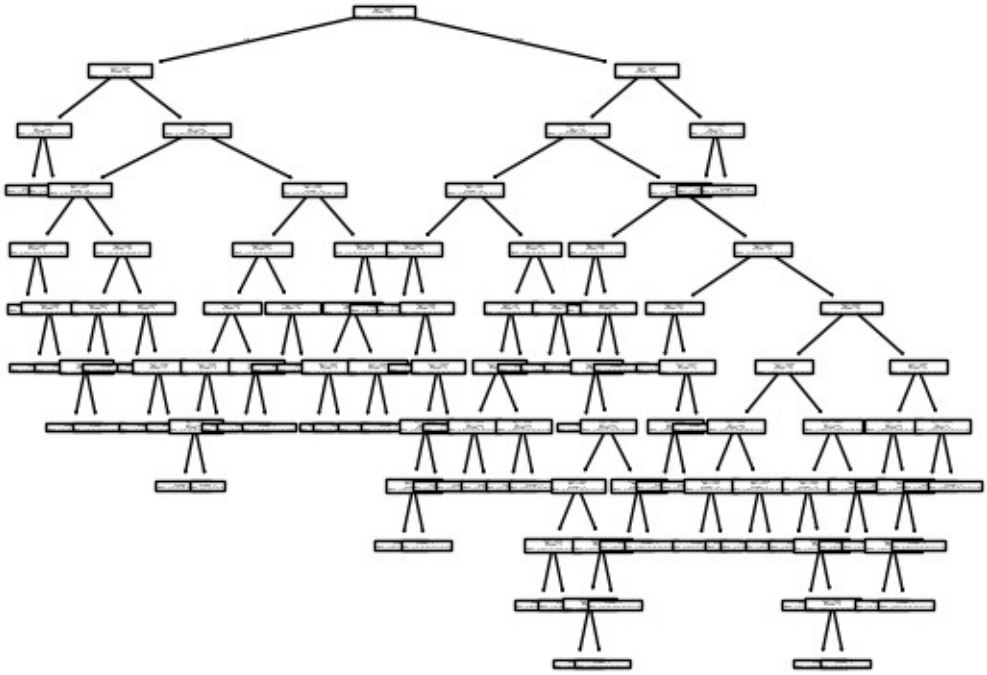
Tras esto usamos lo mismo que en el anterior apartado solo cambiando los parámetros que queremos usar. Tras ejecutar el código nos encontramos con lo siguiente:

```
Mejor parámetros: {'ccp_alpha': 0.0, 'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': None, 'max_features': 10, 'min_samples_leaf': 1, 'min_samples_split': 2, 'splitter': 'random'}

Mejor precisión:  0.742483660130719

Cantidad de predicciones mal hechas de 43 valores : 13
```

Ademas, muestro el árbol resultante con el siguiente código:

```
plot_tree(treeMejor)

plt.show()
```

El cual nos da el siguiente resultado:



Support Vector Machine

Este caso es muy parecido al anterior, en el que tenemos muchos posibles hiperparametros donde decidir. Yo me he quedado con estos:

```
parametrosSVC = {
    "kernel" : ['poly'],
    "gamma" : ['scale', 'auto'],
    "degree" : [25,30,35],
    "shrinking": [True, False],
    "probability": [True, False],
    "decisión_function_shape": ['ovo', 'ovr']
}
```

Donde tenemos un solo tipo de kernel, ya que he probado varias veces y siempre me salia que ese era el mejor, y si dejaba el resto de kernel tardaba mucho en la ejecución. Ademas, para ese tipo de kernel tenemos un hiperparametro exclusivo “degree”.

Para encontrar cuales son los mejores voy a usar las mismas funciones que en los dos anteriores casos. Así que si ejecutamos el código vemos:

Mejor parámetros: {'decisión_function_shape': 'ovo', 'degree': 25, 'gamma': 'scale', 'kernel': 'poly', 'probability': True, 'shrinking': True}

Mejor precisión: 0.718954248366013

Cantidad de predicciones mal hechas de 43 valores : 17

Conclusión sobre los modelos

Como se ha podido ir viendo, ningún modelo es especialmente bueno, aunque nos quedaríamos con el árbol de decisión o Nearest Neighbors al tener ambos unos resultados muy parecidos (Aunque el árbol da un poco mejor). Esto puede ser resultado de tener un dataset muy pequeño que no permite el analices de datos y patrones de manera efectiva, por lo cual es difícil que cualquier algoritmo tenga un desempeño especialmente bueno.

Gráficas

Para comprobar el rendimiento, he usado 3 tipos de métricas:

- **One-vs-Rest multiclass ROC:** Donde lo que hacemos es calcular la curva roc de la cada una de las clases. Para esto, binarizamos el objetivo de manera que esa clase sea positiva y el resto negativa.
- **micro-averaged OvR:** Donde sumamos la contribución de cada clase para calcular TPR(True positive rate) y FPR(False positive rate). Este es mejor cuando las clases están muy disparejas.
- **macro-averaged OvR:** Donde tenemos un resultado parecido al anterior, pero en este caso primero calculamos el TPR y FPR de cada clase por separado y luego hacemos la media.

Para hacer esto, primero he predicho las probabilidades de que sea cada clase con el conjunto de test y los algoritmos con los parámetros decididos antes:

```
y_score_Gaussian = gnb.predict_proba(x_test)

y_score_vecinos = vecinosMejor.predict_proba(x_test)

y_score_tree = treeMejor.predict_proba(x_test)

y_score_svc = svcMejor.predict_proba(x_test)
```

Y después he binarizado el dataset para poder realizar las métricas OvR:

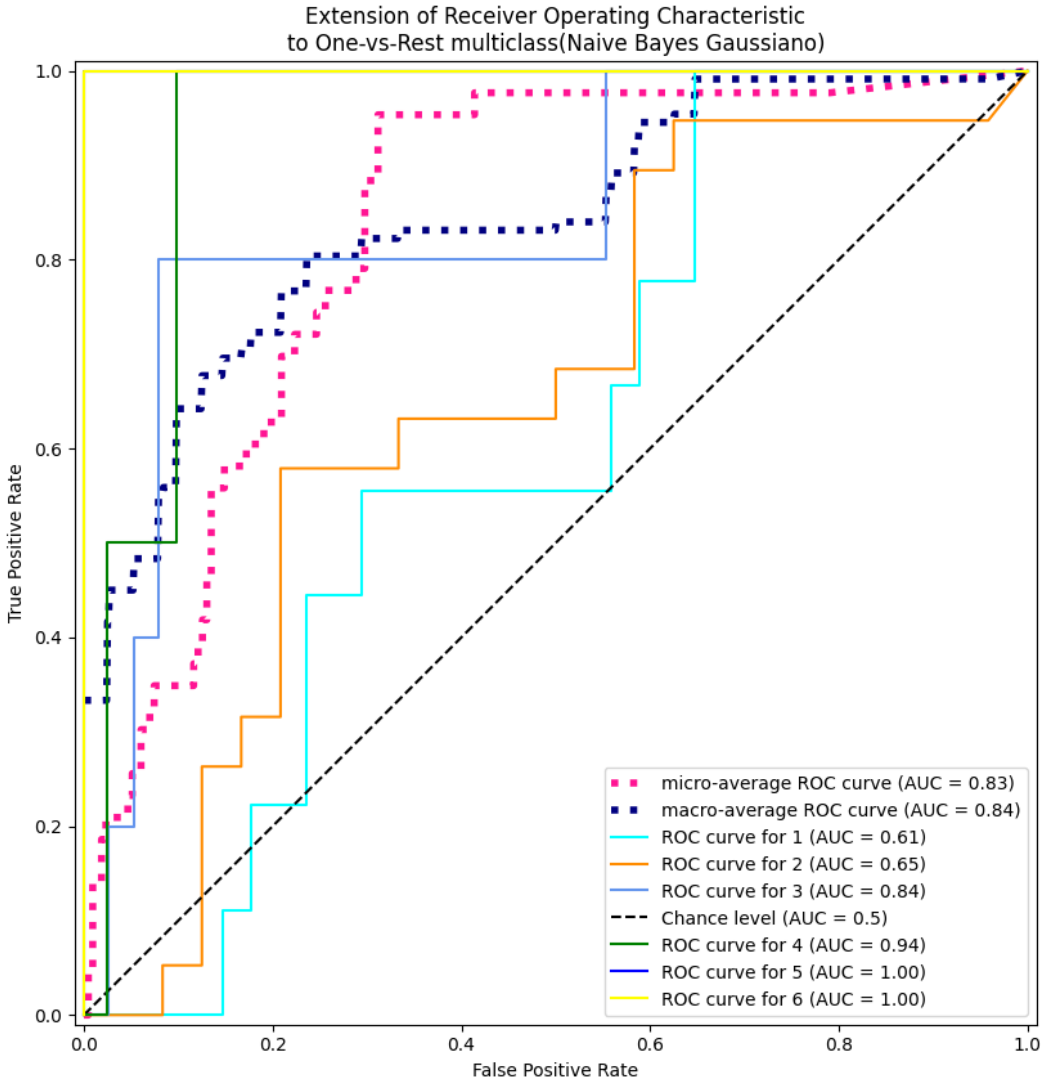
```
target_names = [1,2,3,4,5,6,7]

label_binarizer = LabelBinarizer().fit(y_train)

y_onehot_test = label_binarizer.transform(y_test)

n_classes = y_onehot_test.shape[1]
```

Donde tenemos en target names los nombres de las clases, en este caso he usado numeros al ser muy largos. Luego entreno con los datos de entrenamiento un “LabelBinarizer()”, lo cual nos permite binarizar las clases. Por último, transformo el set de test y guardo el numero de clases.

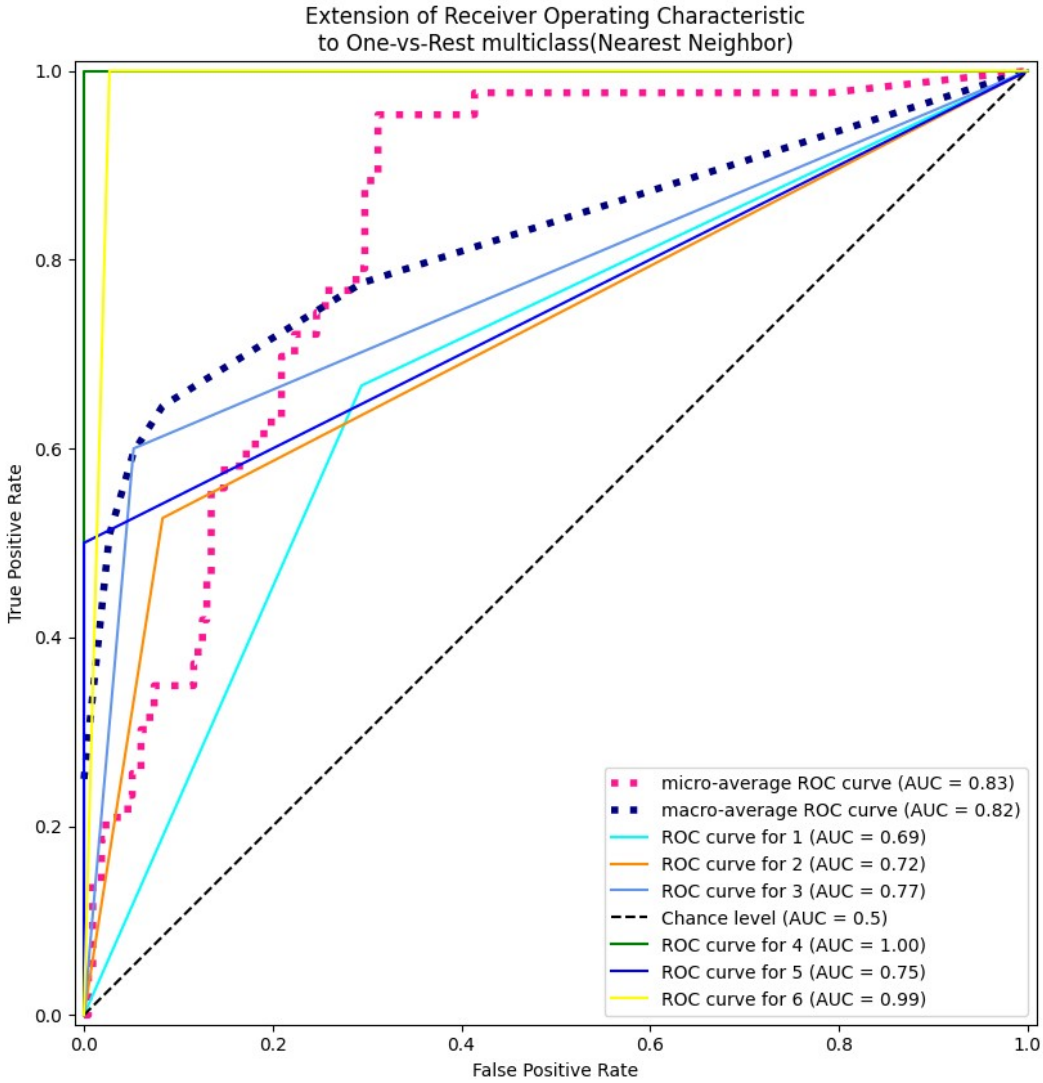
Ahora mismo no hay ninguna clase seleccionada, pero luego ir eligiendo a mano cual quiero usar.

Por ultimo, para mostrar las gráficas he usado “RocCurveDisplay.from_predictions()”, lo cual nos permite crear curvas roc desde las predicciones y los valores binarizados. Y para micro y macro average, calculo ambos valores como indica la documentación y los muestro como un plot normal.

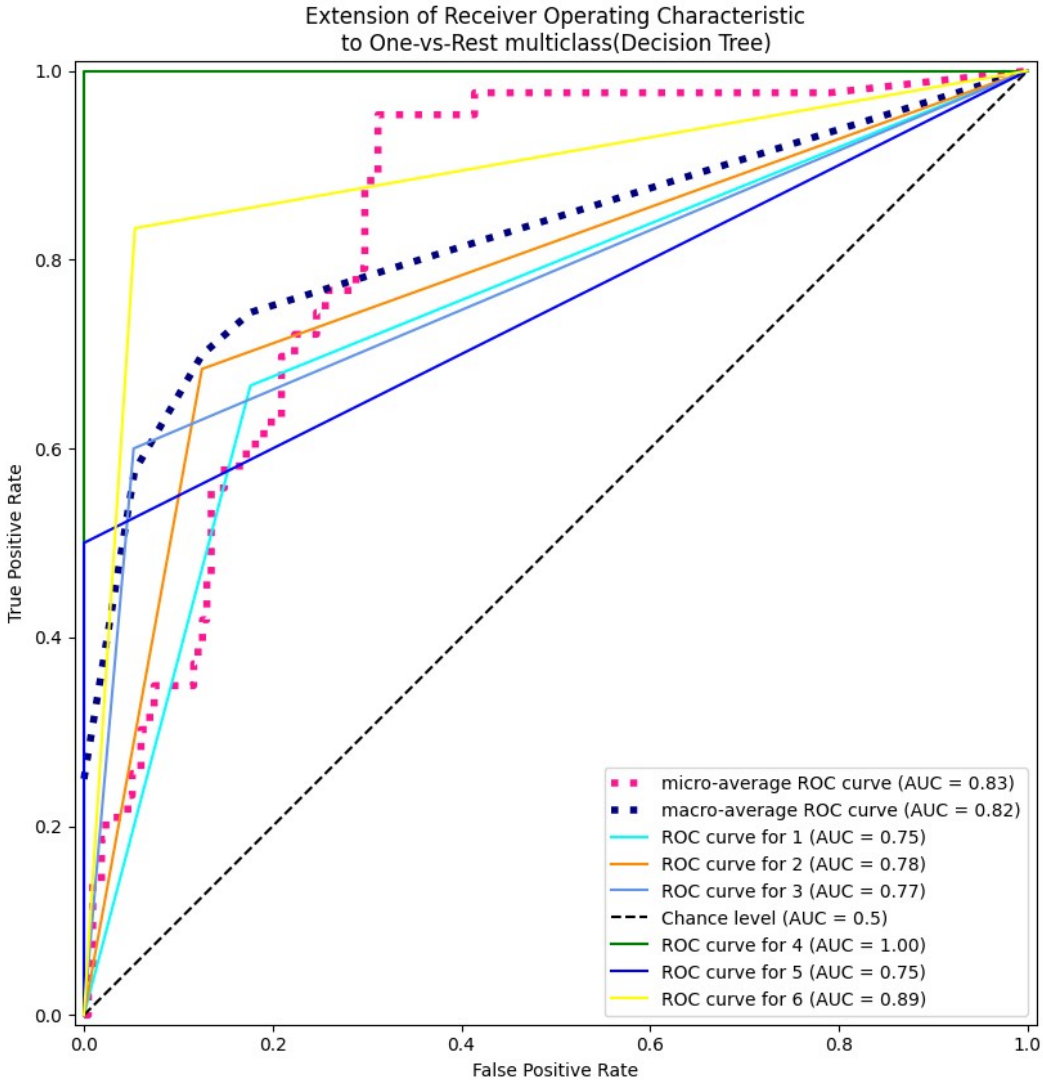
Naive Bayes Gaussiano



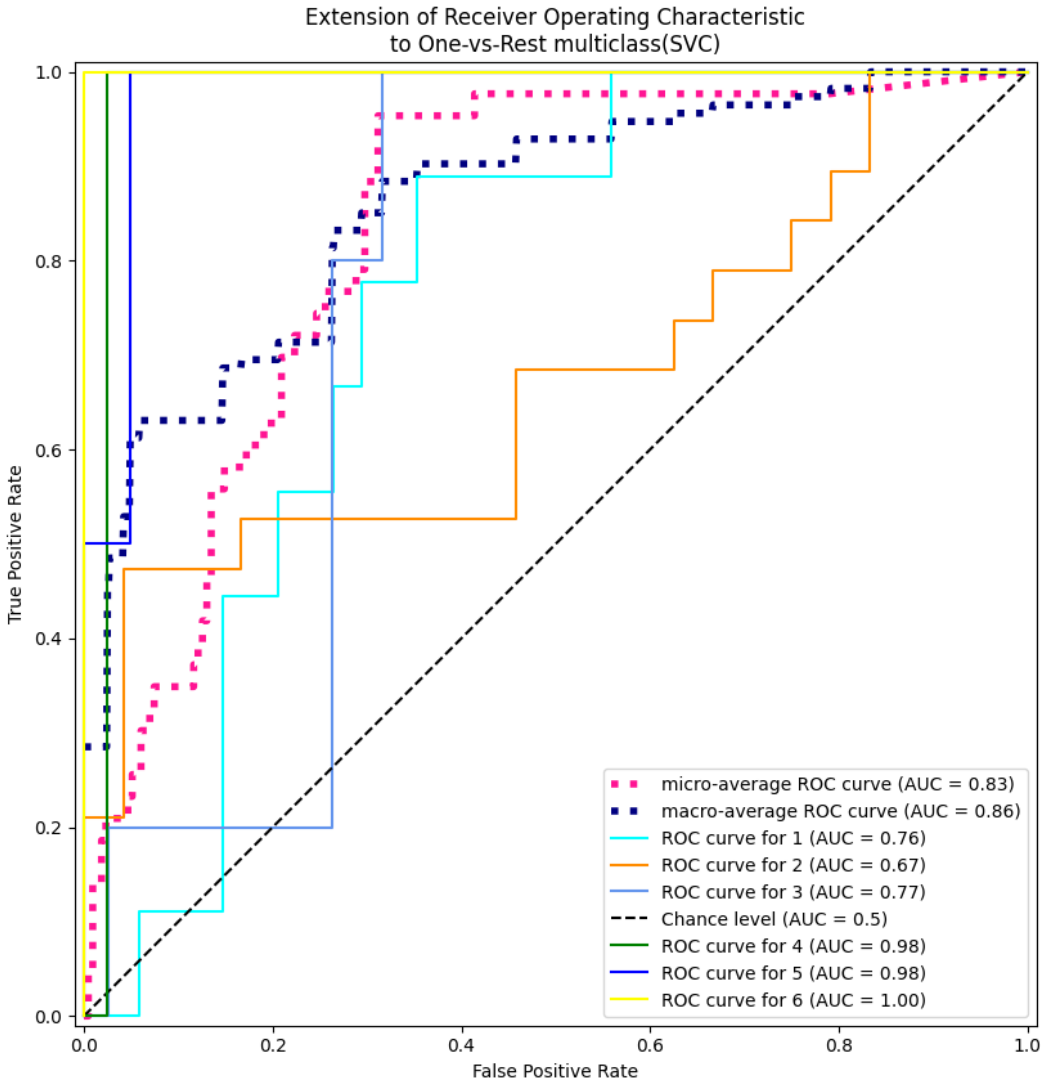
Nearest Neighbor



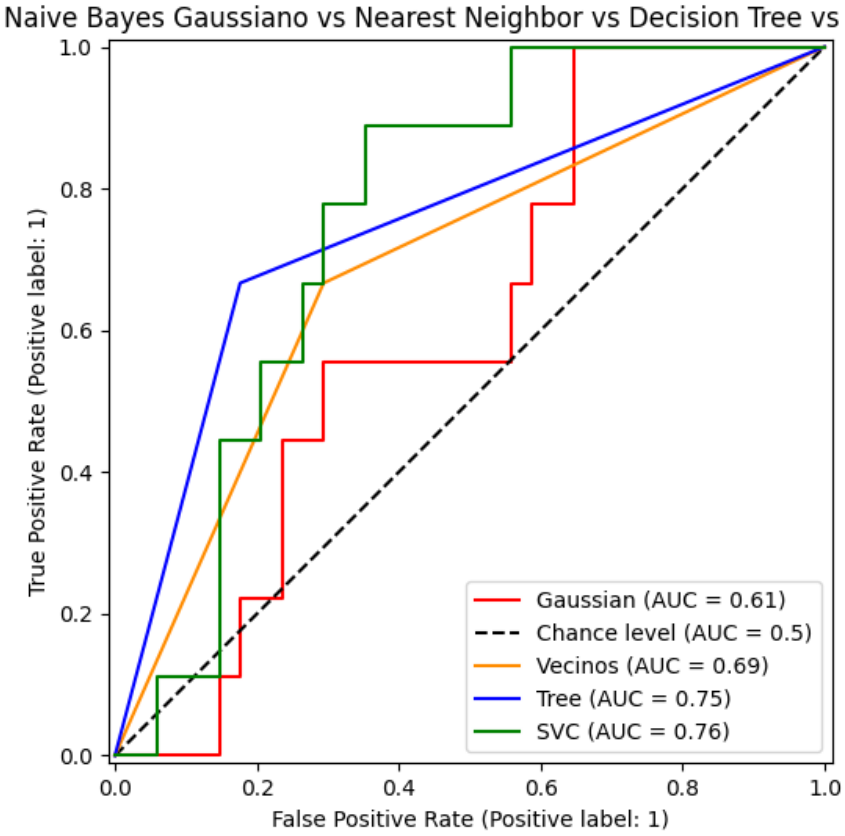
Decision Tree

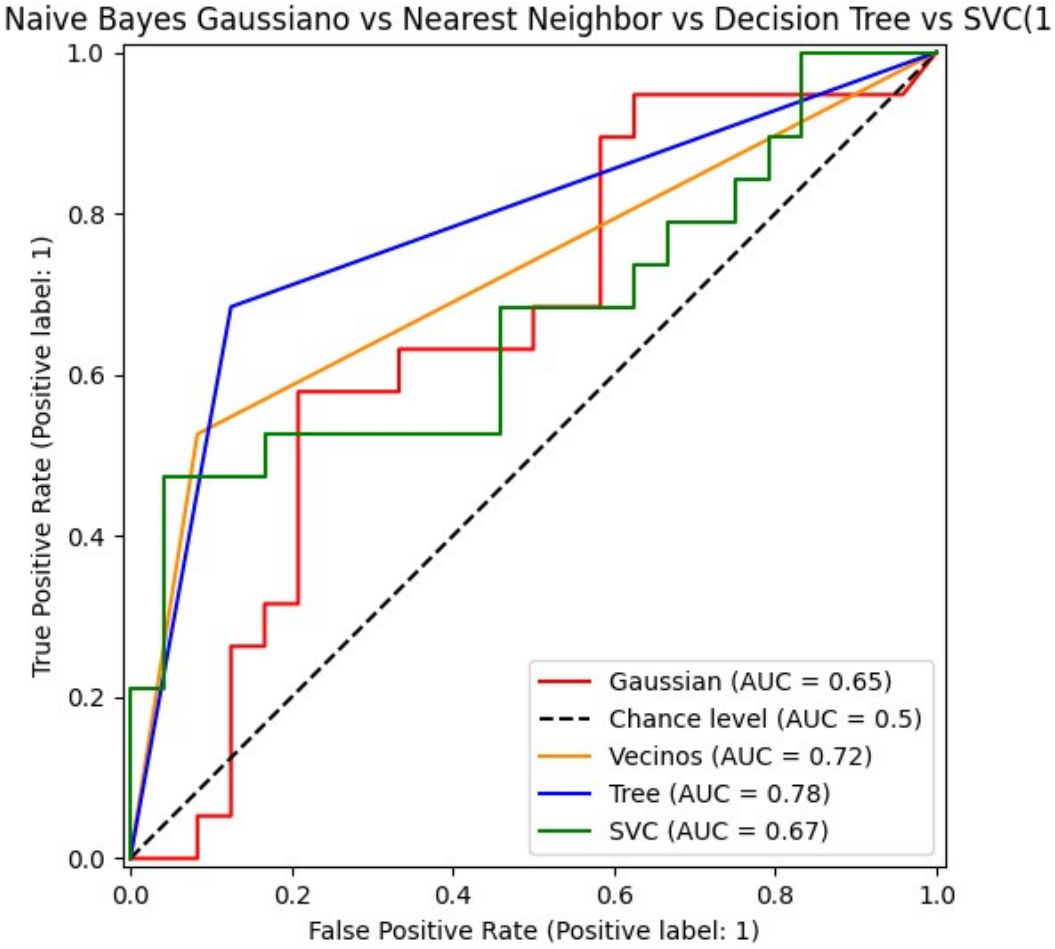


SVC



Por ultimo, para añadir algún gráfico mas, he puesto para cada clase, sus curvas ROC en todos los algoritmos en un mismo gráfico:


Naive Bayes Gaussiano vs Nearest Neighbor vs Decision Tree vs SVC(0
True Positive Rate (Positive label: 1)
False Positive Rate (Positive label: 1)
0.0
0.2
0.4
0.6
0.8
1.0
Gaussian (AUC = 0.61)
Chance level (AUC = 0.5)
Vecinos (AUC = 0.69)
Tree (AUC = 0.75)
SVC (AUC = 0.76)


Naive Bayes Gaussiano vs Nearest Neighbor vs Decision Tree vs SVC(1
True Positive Rate (Positive label: 1)
False Positive Rate (Positive label: 1)
0.0
0.2
0.4
0.6
0.8
1.0
Gaussian (AUC = 0.65)
Chance level (AUC = 0.5)
Vecinos (AUC = 0.72)
Tree (AUC = 0.78)
SVC (AUC = 0.67)

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.svm import SVC
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import RocCurveDisplay
from itertools import cycle
from sklearn.metrics import auc, roc_curve

columnas = ["Id_number", "RI", "Na", "Mg", "Al", "Si", "K", "Ca", "Ba", "Fe", "Type_of_glass"]
#Type of glass: (class attribute)
#      -- 1 building_windows_float_processed
#      -- 2 building_windows_non_float_processed
#      -- 3 vehicle_windows_float_processed
#      -- 4 vehicle_windows_non_float_processed (none in this database)
#      -- 5 containers
#      -- 6 tableware
#      -- 7 headlamps


df = pd.read_csv('data/glass.data', names = columnas)
df = df.drop('Id_number', axis=1)
print("Dataframe:")
print(df)

#Dataframe como X e Y
x = df.iloc[:, :8].values
y = df.iloc[:, 9].values
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=0)
skf = StratifiedKFold(n_splits=10)

#Bayes
print("################### Naive Bayes Gaussiano ###################")
gnb = GaussianNB()
gnb = gnb.fit(x_train, y_train)
y_pred = gnb.predict(x_test)
print("Cantidad de predicciones mal hechas de %d valores : %d" % (x_test.shape[0], (y_test !=
y_pred).sum()))
print("Accuracy "+str((y_test != y_pred).sum()*100/x_test.shape[0])+"%")

#KNeighbours
print("################### Nearest Neighbor ###################")
gsVecinos = GridSearchCV(
    KNeighborsClassifier(),
    param_grid={"n_neighbors":range(1,101)},
    scoring='accuracy',
    cv = skf
)
gsVecinos.fit(x_train, y_train)
print("El mejor numero de vecinos es: ",gsVecinos.best_params_['n_neighbors'])
print("Mejor precision: ", gsVecinos.best_score_)
vecinosMejor = gsVecinos.best_estimator_
y_pred = vecinosMejor.predict(x_test)
print("Cantidad de predicciones mal hechas de %d valores : %d" % (x_test.shape[0], (y_test !=
y_pred).sum()))
```

```
#Decision Tree
print("################## Decision Tree ##################")
parametros = {
    "criterion" : ["gini", "entropy", "log_loss"],
    "splitter" : ["best", "random"],
    "max_depth" : [None],
    "min_samples_split": [2],
    "min_samples_leaf": [1],
    "max_features": [10],
    "class_weight": [None, "balanced"],
    "ccp_alpha": [0.0, 0.1, 0.2, 0.5]
}

gs = GridSearchCV(
    DecisionTreeClassifier(random_state=1),
    param_grid=parametros,
    scoring='accuracy',
    cv = skf
)
gs.fit(x_train, y_train)
print("Mejor parametros:", gs.best_params_)
print("Mejor precisión: ", gs.best_score_)
treeMejor = gs.best_estimator_
y_pred = treeMejor.predict(x_test)
print("Cantidad de predicciones mal hechas de %d valores : %d" % (x_test.shape[0], (y_test !=
y_pred).sum()))

plot_tree(treeMejor)
plt.show()

#Support Vector Machine
print("################## SVC ##################")
parametrosSVC = {
    "kernel" : ['poly'],
    "gamma" : ['scale', 'auto'],
    "degree" : [25,30,35],
    "shrinking": [True, False],
    "probability": [True, False],
    "decision_function_shape": ['ovo', 'ovr']
}
#Mejor 0.7071895424836602
gsSVC = GridSearchCV(
    SVC(random_state=1),
    param_grid=parametrosSVC,
    scoring='accuracy',
    cv = skf
)
gsSVC.fit(x_train, y_train)
print("Mejor parametros:", gsSVC.best_params_)
print("Mejor precisión: ", gsSVC.best_score_)
svcMejor = gsSVC.best_estimator_
y_pred = svcMejor.predict(x_test)
print("Cantidad de predicciones mal hechas de %d valores : %d" % (x_test.shape[0], (y_test !=
y_pred).sum()))

#Graficas
print("################## Graficas ##################")
y_score_Gaussian = gnb.predict_proba(x_test)
y_score_vecinos = vecinosMejor.predict_proba(x_test)
```

```
y_score_tree = treeMejor.predict_proba(x_test)
y_score_svc = svcMejor.predict_proba(x_test)


target_names = [1,2,3,4,5,6,7]
label_binarizer = LabelBinarizer().fit(y_train)
y_onehot_test = label_binarizer.transform(y_test)
n_classes = y_onehot_test.shape[1]
colors = cycle(["aqua", "darkorange", "cornflowerblue", "green", "blue", "yellow", "red", "
brown"])

def plots(y_score, y_onehot_test, n_classes, texto):
    colors = cycle(["aqua", "darkorange", "cornflowerblue", "green", "blue", "yellow", "red", "
brown"])
    #Gaussian Metrica
    fpr, tpr, roc_auc = dict(), dict(), dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_onehot_test[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    fpr_grid = np.linspace(0.0, 1.0, 1000)

    # Interpolate all ROC curves at these points
    mean_tpr = np.zeros_like(fpr_grid)

    for i in range(n_classes):
        mean_tpr += np.interp(fpr_grid, fpr[i], tpr[i])  # linear interpolation

    # Average it and compute AUC
    mean_tpr /= n_classes

    fpr["macro"] = fpr_grid
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

    fpr["micro"], tpr["micro"], _ = roc_curve(y_onehot_test.ravel(), y_score_Gaussian.ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    fig, ax = plt.subplots(figsize=(10, 10))
    plt.plot(
        fpr["micro"],
        tpr["micro"],
        label=f"micro-average ROC curve (AUC = {roc_auc['micro']:.2f})",
        color="deeppink",
        linestyle=":",
        linewidth=4,
    )
    plt.plot(
        fpr["macro"],
        tpr["macro"],
        label=f"macro-average ROC curve (AUC = {roc_auc['macro']:.2f})",
        color="navy",
        linestyle=":",
        linewidth=4,
    )
    for class_id, color in zip(range(n_classes), colors):
        RocCurveDisplay.from_predictions(
            y_onehot_test[:, class_id],
            y_score[:, class_id],
            name=f"ROC curve for {target_names[class_id]}",
```

```
            color=color,
            ax=ax,
            plot_chance_level=(class_id == 2),
        )

    _ = ax.set(
        xlabel="False Positive Rate",
        ylabel="True Positive Rate",
        title="Extension of Receiver Operating Characteristic\nto One-vs-Rest multiclass("
+texto+")",
    )
    plt.tight_layout()
    plt.show()

#Gaussian
plots(y_score_Gaussian, y_onehot_test, n_classes, "Naive Bayes Gaussiano")
#Vecinos
plots(y_score_vecinos, y_onehot_test, n_classes, "Nearest Neighbor")
#Tree
plots(y_score_tree, y_onehot_test, n_classes, "Decision Tree")
#SVC
plots(y_score_svc, y_onehot_test, n_classes, "SVC")

#Metricas comparado un par de valores
def plot_comparar(class_id, y_score_gaussian, y_score_vecinos, y_score_tree, y_score_svc,
y_onehot_test):
    fig, ax = plt.subplots(figsize=(6, 6))
    RocCurveDisplay.from_predictions(
        y_onehot_test[:, class_id],
        y_score_gaussian[:, class_id],
        name="Gaussian",
        color="red",
        plot_chance_level=True,
        ax=ax
    )
    _ = ax.set(
        xlabel="False Positive Rate",
        ylabel="True Positive Rate",
        title="Naive Bayes Gaussiano vs Nearest Neighbor vs Decision Tree vs SVC("+str(class_id)
+")",
    )

    RocCurveDisplay.from_predictions(
        y_onehot_test[:, class_id],
        y_score_vecinos[:, class_id],
        name="Vecinos",
        color="darkorange",
        ax=ax
    )

    RocCurveDisplay.from_predictions(
        y_onehot_test[:, class_id],
        y_score_tree[:, class_id],
        name="Tree",
        color="blue",
        ax=ax
    )

    RocCurveDisplay.from_predictions(
        y_onehot_test[:, class_id],
        y_score_svc[:, class_id],
```

```
        name="SVC",
        color="green",
        ax=ax
    )
    plt.show()

plot_comparar(1, y_score_Gaussian, y_score_vecinos, y_score_tree, y_score_svc, y_onehot_test)
plot_comparar(0, y_score_Gaussian, y_score_vecinos, y_score_tree, y_score_svc, y_onehot_test)
```