#!/usr/bin/env python

# coding: utf-8

# # \*\*Lab 3.1: Machine Learning (Classification)\*\*

#

# <hr>

#

# ## \*\*1. Introduction\*\*

# Once we have analyzed, cleaned, and visualized our dataset, we can move on to the learning phase. Before we begin, it is necessary to identify the type of problem we are facing in order to select the most suitable methods or models.

#

# The following diagram illustrates the most common types of problems in the field of machine learning:

#

# <center><img src="ML\_Diagram.png" alt="diagram" width="1000"/></center>

#

# As you already know, problems that require \*\*machine learning\*\* to be solved are those where we do not know the \*formula\* that allows us to transform the input into the output. These problems are mainly divided into two types: \*\*supervised\*\* and \*\*unsupervised\*\*.

#

# In this course, we will focus on supervised learning problems, where we aim to predict either one or more classes (\*\*classification\*\*) or one or more numerical values (\*\*regression\*\*).

#

# Remember that in order to solve supervised learning problems, we always need \*\*labeled data\*\*, meaning data where we already know the expected output or correct label for a given input. These labeled examples will be used by the model to try to learn that \*unknown formula\* during training.

#

# We will begin by exploring \*\*classification problems\*\*, their characteristics, evaluation metrics, techniques, and how they are used to make predictions.

#

# ### \*\*Objective\*\*

# In this practice, you will learn how to solve classification problems using different models and how to evaluate their performance.

# <hr>

#

# ## \*\*2. Problem Definition\*\*

#

# To begin, we will attempt to create a model capable of solving a \*\*binary classification\*\* problem.

#

# In this case, we need to \*\*create a model that, given the time (in seconds) of the 3 sectors of an \*Aston Martin\* driver, predicts whether the time was set by \*Alonso\* or not (\*Stroll\*)\*\*.

#

# We will reload our data and generate the necessary dataset to solve the problem.

# In[78]:

import pandas as pd

data = pd.read\_pickle('https://raw.githubusercontent.com/AIC-Uniovi/Sistemas-Inteligentes/refs/heads/main/datasets/f1\_23\_monaco.pkl')

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Create the variable <code>data\_aston</code> with a DataFrame that contains only the data of that team and the columns "Sector1Time", "Sector2Time", "Sector3Time", and "Driver". Transform the sector columns from timedelta to seconds using <code>.dt.total\_seconds()</code>.

# </div>

# In[79]:

# Your code here

data\_aston=data[data['Team']=='Aston Martin'][['Driver', 'Sector1Time', 'Sector2Time', 'Sector3Time']]

data\_aston['Sector1Time']=data\_aston['Sector1Time'].dt.total\_seconds()

data\_aston['Sector2Time']=data\_aston['Sector2Time'].dt.total\_seconds()

data\_aston['Sector3Time']=data\_aston['Sector3Time'].dt.total\_seconds()

data\_aston

# We will modify the dataset to transform it into a binary classification problem. As you may recall, in this type of problem, the model predicts \*\*a single value\*\* that indicates the probability between zero ($0\%$) and one ($100\%$) that the given input belongs to the \*\*positive class\*\*.

#

# In our case, the \*\*positive class\*\* will be \*"Alonso"\*. Therefore, if the model predicts a $1$, it indicates that the given sector times belong to Alonso with $100\%$ probability.

#

# If the model's prediction is $0$ or less than $0.5$, it indicates that the lap was not set by Alonso and therefore belongs to Stroll.

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Create the column <code>Class</code> within the DataFrame <code>data\_aston</code> so that it equals zero whenever the driver is not Alonso and 1 otherwise.

# </div>

# In[80]:

# Your code here

#data\_aston['Class']=0

#data\_aston['Class'].loc[data\_aston['Driver'] == 'ALO'] = 1

data\_aston['Class'] = data\_aston['Driver'].apply(lambda x: 1 if x == 'ALO' else 0)

data\_aston

# <hr>

#

# ## \*\*3. Baseline\*\*

#

# Once we have a labeled dataset, where each entry (Sector1Time, Sector2Time, and Sector3Time) is associated with an expected output (Class), we can proceed to create a model to solve the binary classification problem.

#

# As explained in theory class, there are simple models that can provide very good performance without the need for more complex solutions. These models are called \*baselines\* and serve as a reference or lower bound. If a baseline performs better than a much more complex model, something is going wrong.

#

# In classification problems, there are three main ones:

#

# \* \*\*Random:\*\* Predicts a 0 or 1 randomly without considering the input variables.

# \* \*\*Zero-R:\*\* Predicts the majority class, that is, it analyzes the "Class" column of the dataset and, if the most frequent class is 1, it always predicts 1 for any input. As you can see, it doesn't use the input variables at all.

# \* \*\*One-R:\*\* This model selects \*\*one\*\* input variable, the one that offers the best classification possible by itself. It is designed for categorical input variables.

# <div class="alert alert-block alert-warning">

# <strong>NOTE:</strong> We will use <i>Random</i> and <i>Zero-R</i> as baselines for our problem. <i>One-R</i> is not possible, as our input variables are numeric.

# </div>

# ### \*\*Random Baseline\*\*

# We will create a model that generates random predictions (0 or 1) regardless of the input variables.

#

# The implementation is very simple: we just need to generate a random list of zeros and ones. Then, we will count how many times the model has made the correct prediction.

# In[81]:

import random

# Set a seed so that the same random values are generated every time

seed = 2533

random.seed(seed)

# Create a list with as many values (0 or 1) as rows in our dataset

random\_values = [random.choice([0, 1]) for \_ in range(len(data\_aston))]

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> How many times has the model been correct (in percentage)? To calculate this, you need to count how many times the random model predicted a 1 and it was actually a 1, and how many times it predicted a 0 and it was actually a 0.

# </div>

# In[82]:

# Your code here

l=data\_aston['Class'].values

correct\_predicitions=0

for i in range(len(data\_aston)):

if l[i]==random\_values[i]:

correct\_predicitions=correct\_predicitions+1

accuracy = (correct\_predicitions / len(data\_aston)) \* 100

print(f"The accuracy is: {accuracy:.2f} %")

# <div class="alert alert-block alert-success">

# The value you just obtained is one of the most popular <strong>classification</strong> <strong>metrics</strong>, called <strong>accuracy</strong>. It simply counts the number of correct predictions made by the model over the maximum possible.

# </div>

# ### \*\*Zero-R\*\*

# Next, we will obtain the result for the \*Zero-R\* baseline.

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> What is the majority class of the dataset? Considering this, what will be the <strong>accuracy</strong> of <i>Zero-R</i>?

# </div>

# In[83]:

# Your code here

list0=data\_aston['Class'].value\_counts()

majority\_class = list0.idxmax()

print(f"The Majority is: {majority\_class}")

accuracyzero= list0[1]/(len(l))\*100

print(f"Accuracy of Zero-R is: {accuracyzero:.2f}%")

# ## \*\*4. Model Evaluation\*\*

#

# At this point, we \*\*already have two models\*\* capable of solving our binary classification problem: the \*\*Random\*\* model and the \*\*Zero-R\*\* model. Additionally, we know that these models have an accuracy on \*\*resubstitution\*\* of approximately $35\%$ and $60\%$, respectively.

#

# <div class="alert alert-block alert-warning">

# <strong>Resubstitution:</strong> The process by which a model is <u>trained and evaluated using the same dataset</u>. This procedure can provide an initial estimate of model performance, though <strong>it does not reflect its generalization ability</strong>.

# </div>

#

# It is important to remember that when we create a machine learning model, we aim for it to not only perform well with known data but \*\*to be able to make predictions on new, unseen data\*\*.

#

# With this in mind, we can see that the evaluation we just conducted on our models is not entirely appropriate. What we've measured is how well they predict examples they already know, but not their ability to make predictions on future data, which is what really matters.

#

# ### \*\*Strategies\*\*

# To address this issue, rather than evaluating the model on the same data it learned from, we will evaluate it on a part of the dataset that we will have previously separated (test set).

#

# This subset is usually created by randomly selecting a percentage (typically $20\%$) of examples from the original dataset and will be used solely to evaluate the model's performance, not for training. By doing this, we \*"simulate"\* future unseen cases, and if the model has good accuracy on this set, we will know that it is genuinely good.

#

# This strategy is called \*\*Simple Validation\*\*, but there are many other strategies:

#

# - \*\*Cross-validation\*\*: Involves splitting the data into several subsets (\*folds\*), training the model on some of them, and evaluating it on the remaining ones, repeating the process several times. This helps provide a more stable estimate of model performance.

# - \*\*Validation with validation set\*\*: A third subset, distinct from the test set, is used to adjust the model's hyperparameters and prevent overfitting. This is very typical when using neural networks.

#

# To get a more realistic evaluation of our models' performance, we will apply \*\*simple validation\*\* and recheck their accuracy, but this time on the test set.

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Split the <code>data\_aston</code> dataset into training and test sets (80% and 20%) and store the subsets in <code>data\_aston\_train</code> and <code>data\_aston\_test</code>. Use the <code><a href="https://scikit-learn.org/stable/modules/generated/sklearn.model\_selection.train\_test\_split.html">train\_test\_split()</a></code> function from the <i>scikit-learn</i> library, which you need to install in your conda environment first.

# <hr>

# <strong>Set the function's seed so that it always performs the same split (random\_state=2533)</strong>

# </div>

# In[84]:

# Your code here

from sklearn.model\_selection import train\_test\_split

data\_aston\_train, data\_aston\_test = train\_test\_split(data\_aston, test\_size=0.2, random\_state=2533)

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> What is the accuracy of the Random model now, over the test set? Remember to set the seed again.

# </div>

# In[85]:

#Your code here

seed = 2533

random.seed(seed)

random\_values\_test = [random.choice([0, 1]) for \_ in range(len(data\_aston\_test))]

list\_test=data\_aston\_test['Class'].values

correct\_predicitions\_test=0

for i in range(len(data\_aston\_test)):

if list\_test[i]==random\_values\_test[i]:

correct\_predicitions\_test=correct\_predicitions\_test+1

accuracy\_test = (correct\_predicitions\_test / len(data\_aston\_test)) \* 100

print(f"The accuracy is: {accuracy\_test:.2f} %")

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> What is the accuracy of the Zero-R model now, over the test set?

# <hr>

# Remember that you need to recalculate the majority class using the new training set (this is the only data your model should know) and use it to evaluate the model on the test set.

# </div>

# In[86]:

# Your code here

list0\_test=data\_aston\_test['Class'].value\_counts()

majority\_class\_test = list0\_test.idxmax()

print(f"The Majority is: {majority\_class\_test}")

accuracyzero\_test= list0\_test[1]/(len(data\_aston\_test))\*100

print(f"Accuracy of Zero-R is: {accuracyzero\_test:.2f}%")

# As you have seen, the models have obtained values very close to $50\%$ accuracy on the test set, which is far from ideal.

#

# <div class="alert alert-block alert-warning">

# <strong>Note:</strong> The worst possible binary classification model will have an accuracy of 50%, not 0%. A model with 0% accuracy would be predicting everything correctly, but its predictions would need to be inverted.

# </div>

# <hr>

#

# ## \*\*5. Scikit-Learn\*\*

#

# The `Scikit-learn` library we used earlier contains a wide range of models and tools that will be useful for solving machine learning problems.

#

# For example, the baselines we just implemented are already incorporated in the `DummyClassifier()` class.

#

# This class has the `strategy` parameter, which allows us to select the desired baseline and can take, among others, the following values:

# - \*\*uniform\*\*: Equivalent to our \*Random\* baseline.

# - \*\*most\_frequent\*\*: Equivalent to \*Zero-R\* baseline.

#

# In the following code, we can see how to use this class:

# In[87]:

from sklearn.dummy import DummyClassifier

# Create the models

baseline\_random = DummyClassifier(strategy = 'uniform', random\_state = seed)

baseline\_zeror = DummyClassifier(strategy = 'most\_frequent')

# Once the models are created, the next step is to train them, and for that, we need to provide the training data using the `fit()` method, specifically the $X$ and $Y$.

#

# <div class="alert alert-block alert-warning">

# <strong>Note:</strong> In supervised learning, we refer to X as the independent variables or <i>inputs</i> and Y as the dependent variables or <i>outputs</i>.

# </div>

#

# As you know, the goal of the model is to \*learn\* the relationship between $X$ and $Y$ in order to predict $Y$ from new, unknown $X$ values.

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Create the variables X and Y to train the models. Remember that we have split the dataset into two parts.

# </div>

# In[88]:

# Your code here

X=data\_aston\_train[['Sector1Time', 'Sector2Time', 'Sector3Time']]

Y=data\_aston\_train[['Class']]

#x rest of data

#y (data\_aston train) class ->target value

# Train the models

baseline\_random.fit(X, Y)

baseline\_zeror.fit(X, Y)

# Once the model is trained, we can make predictions to evaluate its performance with data that was not seen during training. In `scikit-learn`, this is done using the `predict()` function.

#

# <div class="alert alert-block alert-warning">

# <strong>Note:</strong> This function only takes X (not Y) and returns the predicted Y values.

# </div>

# In[89]:

X\_test = data\_aston\_test[['Sector1Time', 'Sector2Time', 'Sector3Time']]

Y\_test = data\_aston\_test[['Class']]

# Make predictions

pred\_random = baseline\_random.predict(X\_test)

pred\_zeror = baseline\_zeror.predict(X\_test)

# Print results

print(pred\_random)

print(pred\_zeror)

# Another advantage of this library is that it includes many metrics within the `metrics` package, so we can easily obtain the accuracy.

# In[90]:

from sklearn import metrics

print(metrics.accuracy\_score(Y\_test, pred\_random))

print(metrics.accuracy\_score(Y\_test, pred\_zeror))

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Obtain the confusion matrices for the models using <a href="https://scikit-learn.org/stable/api/sklearn.metrics.html">the method implemented in the library</a>. Also, obtain the <i>Precision, Recall, and F1</i> scores using the methods or formulas.

# </div>

#

# <div style="width:800px;background:white;padding:10px">

# <img src="https://i.imgur.com/7WwY9bZ.jpeg" style="margin-bottom:10px"> </img>

# </div>

# In[91]:

"""Y real Y predict

------- --------

1 0

0 0 --> true negative

1 1 --> true positive"""

# In[92]:

# Your code here

from sklearn.metrics import f1\_score, precision\_score, recall\_score

conf\_matrix\_random= metrics.confusion\_matrix(Y\_test, pred\_random)

print(conf\_matrix\_random)

conf\_matrix\_zeror= metrics.confusion\_matrix(Y\_test, pred\_zeror)

print(conf\_matrix\_zeror)

precision\_random = precision\_score(Y\_test, pred\_random)

recall\_random = recall\_score(Y\_test, pred\_random)

f1\_random = f1\_score(Y\_test, pred\_random)

precision\_zeror = precision\_score(Y\_test, pred\_zeror)

recall\_zeror = recall\_score(Y\_test, pred\_zeror)

f1\_zeror = f1\_score(Y\_test, pred\_zeror)

print("")

print(f"Precision Random: {precision\_random}")

print(f"Recall Random: {recall\_random}")

print(f"F1 Score Random: {f1\_random}")

print("")

print(f"Precision Zeror: {precision\_zeror}")

print(f"Recall Zeror: {recall\_zeror}")

print(f"F1 Score Zeror: {f1\_zeror}")

# <hr>

#

# ## \*\*6. Other Models\*\*

# As you have seen in the theory class, in addition to the baselines we have used, there are many other different models to solve classification problems. Some of them can be:

#

# \* \*\*Logistic Regression:\*\* A "linear" model that uses a logistic function (sigmoid) to predict the probability of belonging to a class.

# \* \*\*K-Nearest Neighbors:\*\* A classifier that assigns an instance to the most frequent class among its k nearest neighbors.

# \* \*\*Decision Trees:\*\* A classification model that creates a tree allowing decisions based on the data features (inputs).

# \* \*\*SVM:\*\* A classification algorithm that seeks the optimal hyperplane to maximize the margin between classes.

# \* \*\*Neural Networks:\*\* We will explore them in detail in the upcoming topics.

#

# Just like with the baselines, `scikit-learn` provides us with most of these models already implemented.

# <div class="alert alert-block alert-warning">

# <strong>Note:</strong> Before using other models, it is necessary to carry out a phase of <u>data preprocessing</u> that we had pending: <strong>normalization or standardization</strong> of the data.

# </div>

# As you may remember, this phase ensures that all the <u>inputs</u> to our model (sector times in our case) are in the same range (normalization) or have the same mean and standard deviation (standardization) to facilitate learning.

#

# We haven't done this so far because the previous baselines did not use any input information; one predicted randomly, and the other predicted the most frequent class.

#

# In this case, we will perform standardization (normalization would also be valid). Let's first check the mean and standard deviation of the inputs.

# In[93]:

# Mean and standard deviation of the inputs in our model

X.describe().loc[["mean", "std"]]

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Use the <a href="https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html"><code>StandardScaler()</code></a> from <i>sklearn</i> to standardize the training and test X data. Store the new data in <code>X\_std</code> and <code>X\_test\_std</code>. Remember that the test data is unknown to us, so it should not influence the calculation of the mean and standard deviation.

# </div>

# In[94]:

from sklearn.preprocessing import StandardScaler

standardizer = StandardScaler()

# Your code here

X\_std=standardizer.fit\_transform(X)

X\_test\_std=standardizer.transform(X\_test)

#standarized for everything pq entran las X

#Variables numéricas standarizar ar que estñe entre los mismos valores

#Variables categóricas tb hay que normalizarlo en 1,2,3

# Verify with:

print(X\_std.mean(axis = 0))

print(X\_std.std(axis = 0))

# ### \*\*6.1. Logistic Regression\*\*

#

# We are now ready to train and evaluate new models <u>aiming to improve the baselines' metrics</u>. We will start with \*\*Logistic Regression\*\*.

#

# To use it, we simply need to create an object of the `LogisticRegression()` class, train it with `fit()`, and evaluate it with `predict()`.

# In[95]:

from sklearn.linear\_model import LogisticRegression

# Create, train, and evaluate the model

model\_log = LogisticRegression()

model\_log.fit(X\_std, Y.squeeze()) # The squeeze removes unnecessary dimensions

pred\_log = model\_log.predict(X\_test\_std)

print(metrics.accuracy\_score(Y\_test, pred\_log))

# As you can see, this model achieves better results in terms of accuracy, but not much better.

# ### \*\*6.2. K-Nearest Neighbors\*\*

#

# The next model we can try is the `KNeighborsClassifier()`, which is used exactly the same way as the others.

# In[96]:

from sklearn.neighbors import KNeighborsClassifier

# Create, train, and evaluate the model

model\_knn = KNeighborsClassifier(n\_neighbors = 3)

model\_knn.fit(X\_std, Y.squeeze())

pred\_knn = model\_knn.predict(X\_test\_std)

print(metrics.accuracy\_score(Y\_test, pred\_knn))

# The results in terms of accuracy with this model are very close to 100%, making it the best among those evaluated.

# ### \*\*6.3. Decision Trees\*\*

#

# Next, we will analyze the performance of a decision tree in solving this task. The class within the library is called `DecisionTreeClassifier()` and it has numerous hyperparameters, with one of the most notable being `max\_depth`, which allows setting the maximum depth of the tree and thus controlling \*\*overfitting\*\*.

# In[97]:

from sklearn.tree import DecisionTreeClassifier

# Create, train, and evaluate the model

model\_tree = DecisionTreeClassifier(random\_state = seed, max\_depth = 2)

model\_tree.fit(X\_std, Y.squeeze())

pred\_tree = model\_tree.predict(X\_test\_std)

print(metrics.accuracy\_score(Y\_test, pred\_tree))

# The performance is similar to the previous model, but this one has the advantage of allowing us to see its internal workings, i.e., the tree it has created. To do this, we will need to install a new library.

# In[98]:

get\_ipython().system(' pip install pydotplus')

# In[99]:

from sklearn.tree import export\_graphviz

from IPython.display import Image

import pydotplus

# Export the resulting tree to DOT format

dot\_data = export\_graphviz(decision\_tree = model\_tree, feature\_names = X.columns, class\_names = ['Stroll', 'Alonso'], filled = True)

graph = pydotplus.graph\_from\_dot\_data(dot\_data)

Image(graph.create\_png())

# ### \*\*6.4. Support Vector Machines (SVM)\*\*

#

# The last model we are going to try is the SVM, implemented in the `SVC()` class. We will start with a \*\*linear kernel\*\*.

# In[ ]:

from sklearn.svm import SVC

# Create an SVM with a linear kernel, train, and evaluate

model\_svm = SVC(kernel = 'linear')

model\_svm.fit(X\_std, Y.squeeze())

pred\_svm = model\_svm.predict(X\_test\_std)

print(metrics.accuracy\_score(Y\_test, pred\_svm))

# The accuracy is not very high, so it's possible that the classes are not \*\*linearly separable\*\*.

#

# To improve performance, we will now try a \*\*polynomial kernel\*\* of degree 2.

# In[ ]:

# Create an SVM with a polynomial kernel of degree 2 and independent term 1. Train and evaluate

model\_svm\_p = SVC(kernel = 'poly', degree = 2, coef0 = 1)

model\_svm\_p.fit(X\_std, Y.squeeze())

pred\_svm\_p = model\_svm\_p.predict(X\_test\_std)

print(metrics.accuracy\_score(Y\_test, pred\_svm\_p))

# As you can see, the non-linearity of this model allows it to solve the problem with a precision identical to the best models.

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Complete the function <code>train\_and\_eval()</code> for the remaining models, including both versions of the SVM. It uses the function <code>train\_and\_eval\_model()</code> implemented below.

# </div>

# In[ ]:

def train\_and\_eval\_model(model\_name, model, X\_std, Y, X\_test\_std, Y\_test):

# Train the model

model.fit(X\_std, Y.squeeze())

# Predictions

Y\_train\_pred = model.predict(X\_std)

Y\_test\_pred = model.predict(X\_test\_std)

# Calculate metrics for training data

tr\_accuracy = metrics.accuracy\_score(Y, Y\_train\_pred)

tr\_precision = metrics.precision\_score(Y, Y\_train\_pred, zero\_division = 0)

tr\_recall = metrics.recall\_score(Y, Y\_train\_pred)

tr\_f1 = metrics.f1\_score(Y, Y\_train\_pred)

# Calculate metrics for test data

tst\_accuracy = metrics.accuracy\_score(Y\_test, Y\_test\_pred)

tst\_precision = metrics.precision\_score(Y\_test, Y\_test\_pred, zero\_division = 0)

tst\_recall = metrics.recall\_score(Y\_test, Y\_test\_pred)

tst\_f1 = metrics.f1\_score(Y\_test, Y\_test\_pred)

return (model\_name, tr\_accuracy, tr\_precision, tr\_recall, tr\_f1, tst\_accuracy, tst\_precision, tst\_recall, tst\_f1)

# In[ ]:

def train\_and\_eval(X\_std, Y, X\_test\_std, Y\_test):

# Create a list to store the results of each model

all\_results = []

# Random baseline

baseline\_aleatorio = DummyClassifier(strategy = 'uniform', random\_state = seed)

model\_results = train\_and\_eval\_model('Random', baseline\_aleatorio, X\_std, Y, X\_test\_std, Y\_test)

all\_results.append(model\_results)

# Your code here

#Zeror

baseline\_zeror = DummyClassifier(strategy='most\_frequent')

model\_results = train\_and\_eval\_model('ZeroR', baseline\_zeror, X\_std, Y, X\_test\_std, Y\_test)

all\_results.append(model\_results)

# Logistic Regression

model\_log = LogisticRegression()

model\_results = train\_and\_eval\_model('Logistic Regression', model\_log, X\_std, Y, X\_test\_std, Y\_test)

all\_results.append(model\_results)

# K-Nearest Neighbors

model\_knn = KNeighborsClassifier(n\_neighbors=3)

model\_results = train\_and\_eval\_model('K-Nearest Neighbors', model\_knn, X\_std, Y, X\_test\_std, Y\_test)

all\_results.append(model\_results)

# Decision Tree

model\_tree = DecisionTreeClassifier(random\_state=seed, max\_depth=2)

model\_results = train\_and\_eval\_model('Decision Tree', model\_tree, X\_std, Y, X\_test\_std, Y\_test)

all\_results.append(model\_results)

# SVM with linear kernel

model\_svm = SVC(kernel='linear')

model\_results = train\_and\_eval\_model('SVM (Linear Kernel)', model\_svm, X\_std, Y, X\_test\_std, Y\_test)

all\_results.append(model\_results)

# SVM with Ppolinomial kernel

model\_svm\_poly = SVC(kernel='poly')

model\_results = train\_and\_eval\_model('SVM (Polinomial Kernel)', model\_svm\_poly, X\_std, Y, X\_test\_std, Y\_test)

all\_results.append(model\_results)

# Print the resulting dataframe

multi\_index = pd.MultiIndex.from\_tuples([ ('Model', 'Name'), ('Train', 'Accuracy'), ('Train', 'Precision'), ('Train', 'Recall'), ('Train', 'F1'), ('Test', 'Accuracy'), ('Test', 'Precision'), ('Test', 'Recall'), ('Test', 'F1') ])

all\_results = pd.DataFrame(all\_results, columns = multi\_index)

display(all\_results)

train\_and\_eval(X\_std, Y, X\_test\_std, Y\_test)

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Which model do you think is the best? Why?

# </div>

# In[ ]:

# Your response here

#acuracy and preciosion para binaria

#multietiqueta 1,2,3 ... recall y f1

#qu clases el modilo se confnude?

# <div class="alert alert-block alert-info">

# <b>Exercise:</b> Create a model capable of predicting, based on the 3 sectors of a driver, whether the tire used for the lap is for wet conditions ("INTERMEDIATE" or "WET") or not. Perform all necessary preprocessing and try all the classification models we have seen so far using the previous function.

# </div>

# In[ ]:

# Your code here

#create columns named tire, and if its intermediate or wet put 1 if not 0 and do

# In[101]:

data\_lluvia=data[["Sector1Time", "Sector2Time", "Sector3Time", "Compound"]].copy()

data\_lluvia["Sector1Time"]=data\_lluvia["Sector1Time"].dt.total\_seconds()

data\_lluvia["Sector2Time"]=data\_lluvia["Sector2Time"].dt.total\_seconds()

data\_lluvia["Sector3Time"]=data\_lluvia["Sector3Time"].dt.total\_seconds()

data\_lluvia["Class"]=data\_lluvia["Compound"].apply(lambda x: 1 if (x=="WET" or x=="INTERMEDIATE") else 0)

data\_lluvia\_train, data\_lluvia\_test=train\_test\_split(data\_lluvia, test\_size=0.2, random\_state=seed)

X\_train = data\_lluvia\_train[['Sector1Time', 'Sector2Time', 'Sector3Time']]

Y\_train = data\_lluvia\_train['Class']

# In[103]:

X\_test = data\_lluvia\_test[['Sector1Time', 'Sector2Time', 'Sector3Time']]

Y\_test = data\_lluvia\_test['Class']

scaler = StandardScaler()

X\_train\_std = scaler.fit\_transform(X\_train)

X\_test\_std = scaler.transform(X\_test)

train\_and\_eval(X\_train\_std, Y\_train, X\_test\_std,Y\_test)

# In[ ]:

# In[ ]: