**[Analisis de datos](https://docs.google.com/document/d/1i6ll6XBpYLUU38-xXP7oOAPboz2cN4IK/edit)**

* **Procesamiento de datos:** El 80% de éxito del resultado de Machine Learning es todo lo que es el procesamiento de dato.
  + Entender nuestros datos
  + Transformar si no tienen una estructura adecuada
  + Revisar datos faltantes y revisar tipología de las variables
* **Tipos de datos:** Podemos clasificar los datos conforme a la estructura de la información y existen tres formatos básicos:

1. **Datos estructurados:** Corresponde a toda la información que esta identificada conforme a un patron común y definido que representa un modelo. Una estructura optimo para ser analizado. Ej. Modelo relacional, base de datos SQL. (Existe relación entre tablas)
2. **Datos semi-estructurados:** La estructura de los datos esta definida y es conocida, pero no responde a un modelo estricto. Una estructura optima para ser procesado. Ej. JSON, XML. (No existe relación entre tablas)
3. **Datos no estructurados:** No representa un modelo y carece de estructura conocida. Requiere ser transformada para su análisis. Ej. Multimedia, texto libre.

Aproximadamente el 10-15% de los datos que tenemos disponible son de tipo **estructurado o semi estructurado.**

La mayor parte del tiempo dedicada al procesamiento de los datos incluye la extracción de información y la transformación de los datos de un formato a otro.

Ej. Dato no estructurado (Tenemos una imagen) **>** Dato Semi-estructurado (Formato JSON que contiene información extraída de la imagen- Como por ejempla la fecha, nombre de la imagen, metadatos que describen la imagen) > Dato estructurado (Cuando ya tenemos esa información en una tabla definida – Columnas y filas)

**Nota: Los datos estructurados no necesariamente quiere decir que son datos de calidad.**

**Estadistica Descriptiva:**

Se basa en la necesidad de comprender nuestros datos a la perfección. Ver que forma tienen, si tienen sesgo, ver la cantidad de datos por cada tipología, para asi saber que tratamiento darle a cada tipología de dato.

* **Finalidad:** Tener un modelo robusto y fiable.

Trabajaremos en el proceso de:

* Utilizar técnicas principales para aprender de nuestros datos.
* Entender los datos a través de la observación estadística como: Las dimensiones, tipo de datos, distribución de clases, entre otros.
* Trabajar estadística avanzadas en el análisis de un conjunto de datos como desviaciones estándar y sesgo de los datos.
* Entender las relaciones entre los atributos mediante el calculo de correlaciones.

1. **Entender nuestros datos:** 
   1. **Error típico:** Trabajar directamente algoritmos sin conocer nuestros datos.
   2. Una buena cocina de datos (Análisis y procesamiento) nos lleva a obtener mejores y mas confiables resultados, entendiendo nuestro conjunto de datos. Conocer como están estructurados, conocer que parte son buenas de mis datos y que parte son malas, identificar si tiene muchas características aplicar alguna técnica de selección de características.
   3. **Las tareas a realizar son:** 
      1. **Limpieza de datos:** Descubrimos datos corruptos y marcamos o imputamos datos faltantes.
      2. **Transformación de datos:** Descubrir distribuciones como la gaussiana o exponecial que nos genera una idea de que técnica utilizar. Como por Ej. [Escalamiento](https://docs.google.com/document/d/1-Rm8s29CFoC1F8tG0FY0Emmv0ERxSBdd/edit) o transformación de datos.
      3. **Modelo de datos:** A través de la estadística descriptiva obtenemos un idea de que algoritmo poder utilizar. (Consultar: Algoritmo de taxonomía lineal, [modelo](https://www.ibm.com/es-es/think/topics/lasso-regression) [Lasso](https://interactivechaos.com/es/manual/tutorial-de-machine-learning/regresion-lasso), [Elastic](https://www.ibm.com/docs/es/spss-statistics/saas?topic=features-linear-elastic-net-regression) [Net](https://interactivechaos.com/es/manual/tutorial-de-machine-learning/elastic-net):)
   * La función **head()** nos permite ver un resumen de los registros que contiene nuestro conjunto de datos. Esto nos permite ver a grandes rasgos que información contiene el DF, nos puede dar una idea inicial si es un problema de clasificación o de regresión.
   * **Dimensiones del dataset:** Utilizamos la función **.shape** para identificar el nuero de instancias (Registros) y atributos(Columnas).
     + Cuando se tienen muchos atributos(columnas) puede presentarse un problema de dimensionalidad.
     + Muchas instancias (Registros) entonces es conveniente disminuirlas (Capacidad computacional).
   * **Tipo de datos:** La función **.dtypes** nos indica que tipo de dato es cada característica. Es importante por que nos da alguna idea de:
     + Que tipo de análisis podemos utilizar.
     + Tipo de visualizaciones de conjunto de dato
     + Que algoritmo de ML puede ser un bue cantidato

También nos da una idea de que variables debemos de realizar cambio de tipo de dato. De acuerdo a la necesidad

* + **Resumen de los datos:** La función **.describe()** nos permite ver las estadísticas primarias para cada atributo. Este solo nos va a mostrar de manera predeterminada los datos que son de tipo numérico. Como por Ej. La media, el máximo, el mínimo, etc. De cada variable numérica del DF.

**Dependiendo de la importancia de la variable podemos descubrir cosas interesantes, como por Ej.** Si una de las columnas tiene como valor mínimo 0, dada la naturaleza de la columna puede ser normal o sospechoso. Asi mismo con las demás características como el máximo.

Con la desviación estándar (**std**) podemos ver si tenemos mucha varianza en un mismo dato. Dependiendo de si tenemos mucha varianza en una columna o vemos que tiene mas varianza de los normal que las demás, debemos de revisar si nos conviene hacer una estandarización de los datos para eliminar la varianza.

* + **Correlacion:** Se utiliza la función **.corr()** para revisar la relación entre pares de atributos numéricos
    - Los valores superiores a 0.75 o inferiores a -0.75 muestran un alta correlación. Siendo 1 y -1 correlación positiva o negativa completa.
    - Nota: En un análisis de variables para un algoritmo, entre mayor sea la correlación con la variable objetivo, se considerara una variable importante para el modelo. Por lo tanto, la correlacion nos puede ayudar a seleccionar variables para el modelo de acuerdo con su correlación. **Solo variables numéricas**
      * **Cuando existe mucha correlación entre 2 variables independientes, se debe de eliminar una (Se consideran iguales, entre menos corr mejor)**

1. **Visualización de los datos:** La finalidad es detectar valores atípicos o comportamientos que no puedan ayudar a entender mejor el conjunto de datos.
   1. Ademas nos proporciona una idea de las posibles transformaciones de datos que podríamos aplicar en la fase de procesamiento de datos.
   2. Tenemos 2 tipos de visualización:
      1. Visualización univariables: Mostrar atributos o columnas de manera independiente.
      2. Visualización multivariables: Mostrar varios atributos o columnas
2. **Visualización univarieble:**
   1. **Histograma:** Utilizamos la función **.hist()** la cual nos muestra un grafico de barrar con la frecuencia o numero de instancias de cada valor único de la columna que estemos graficando. De esta manera podemos ver como están distribuidos los datos y **valores atípicos que esten fuera del conjunto de datos**.
   2. **Densidad:** Para conocer la densidad de una variable **numérica** podemos utilizar la función **.plot()** con el parámetro **kind=”density”** es como ver un histograma pero en líneas, donde podremos ver la distribución de la variable.
   3. **Boxplot:** Una de las manera de crear este grafico es con la función **.plot()**  con el parámetro **kind=”box”** la cual representa lo siguiente:
      1. La caja captura el 50% medio de los datos (Del 25% al 75%)
      2. La línea del medio de la caja corresponde a la mediana. Si la mediana esta dando mas a un lado y no en todo el centro, puede indicar un sesgo.
      3. Los bordes de los graficos muestra la extensión razonable de los datos
      4. Cualquier punto fuera de los bordes es un buen candidato para los valores atípicos.
3. **Visualizacion Multivariables:**
   1. **Correlacion:** Podemos utilizar la función **heatmap()** para graficar la matriz de correlación de las variables numéricas.

**Caso practico de Analisis de datos.**

1. **Distribución entre clase:** Para ver la distribución entre clases, es decir cuanto hay de cada una de las clases de una columna, podemos utilizar **.groupby()** donde le especificaremos la columna que queremos agrupar mas la función **.size()** la cual nos dará la cantidad de cada uno de los valores.

**Ej.**

#Cantidad de valores por cada registro de la columna

df\_muestra.groupby("country").size()

Esto tambien lo podemos hacer con la función **.value\_counts()**

**Ej.**

df\_muestra["country"].value\_counts().sort\_index()

En este ejemplo le agremos la función **.sort\_index()** para que ordene el resultado por el índice, que en este caso viene siendo la columna “country”

1. **Correlacion entre variables:** Para analizar la correlación entre variables numéricas podemos utilizar la función **.corr()** de esta manera podremos ver que tan dependientes o independientes son las variables entre si.
   1. Para entrenar un buen modelo, entre menos correlacion exista entre las variables independientes mejor será, ya que asi no correríamos el riesgo de caer en overttfiting. **Se considera una alta correlacio cuando el valor esta por encia de 0.75 o -0.75.**
   2. Mientras que la correlación respecto a la variable objetivo, lo ideal es que sea lo mas alta posible. Las variables que menos correlación tenga con la variable objetivo se consideraran de poca importancia para el modelo.

**Ej.**

#Correlacion de las variables numericas

df\_int.corr()

1. **Sesgo de cada atributo:** Para calcular el sesgo de cada atributo podemos utilizar la función **.skew().** El resultado de inclinación muestra una inclinación positiva (derecha) o negativa (izquierda). Los valores mas cercanos a cero muestran menos sesgo.

**Ej.**

#Revisando el sesgo de las vairbales numericas

df\_int.skew()

1. **Densidad:** Corresponde a graficas que se ven como histogramas donde podremos ver la distribución de cada atributo de una manera mas clara y para eso podemos utilizar la función **.plot()** con el parámetro **kind=”density’**

**Ej.**

#Se crea grafica de densidad por cada variable numerica

fig = plt.figure(figsize=(17,17))

ax = fig.gca()

df\_int.plot(ax=ax,kind="density", layout=(5,6), subplots=True, sharex=False)

plt.show

En el ejemplo anterior primero definimos el tamaño de la figura con **plt.figure()** en este caso 17x17 pulgadas. Posterior a eso con la función .**gca()** obtenemos el eje actual de la figura, sobre la cual se va a realizar la grafica.

Finalmente con **.plot()** creamos los graficos para las diferentes variables del DF, donde le indicamos que queremos que sea un grafico de densidad con el parámetro **kind=”density”** con **layout=(5,6)** organiza las graficas en 5 filas y 6 columnas, con **subplots=True** creamos multiples graficas en lugar de una sola y con **sharex=False** indicamos que el eje X no será compartido entre las subgraficas

1. **Boxplots:** Con este tipo de grafico podemos ver si tenemos valores atípicos, identificar la posición del valor medio y también que tan segados pueden estar los datos. Para eso podemos utilizar la funcion de **.plot()** con el parámetro **kind=”box”**

**Ej.**

#Se crea grafica de cajas (boxplot) por cada variable numerica

fig = plt.figure(figsize=(17,17))

ax = fig.gca()

df\_int.plot(ax=ax,kind="box", layout=(5,6), subplots=True, sharex=False)

plt.show

Aplica la misma descripción que en el punto anterior, solo que se cambia el valor del parámetro **kind**

**Ej 2. Esto también lo podemos conseguir utilizando la función de Boxplot de la librería de seaborn.**

#Grafica de caja con seaborn

f, axes = plt.subplots(3,3, figsize=(14,14))

sns.boxplot(x=df\_int["atm\_transfer\_in"], ax=axes[0,0])

sns.boxplot(df\_int["atm\_transfer\_out"], ax=axes[0,1])

sns.boxplot(df\_int["bank\_transfer\_in"], ax=axes[0,2])

sns.boxplot(df\_int["bank\_transfer\_out"], ax=axes[1,0])

Donde primero definimos la figura y la cantidad de filas y columnas (3,3)

**Análisis Multivariables**

1. **Matriz de correlación:** Si bien podemos ver como están correlacionadas las variables utilizando la función **.corr()** lo podemos hacer también de una manera grafica utilizando un mapa de calor utilizando la funcion **.heatmap()**

**Ej.**

#Grafica de correlacion

corr = df\_int.corr() #Se defina la correlacion

fig = plt.figure(figsize=(17,17)) #Se define tamaño de la fig

ax = fig.gca() #Se ptiene el eje actual

sns.heatmap(corr, annot=True, ax=ax) #Se realiza mapa de calor

plt.title("Matriz de correlacion")

En este ejemplo estamos graficando un mapa de calor con la matriz de correlacion previamente definida. 1. Creamos el tamaño de la figura, 2. Optenemos el eje actual de la figura y 3. utilizamos la funcion **.heatmap()** donde le pasamos la matriz a graficar (corr) con el paraetro **annot=True** para que muestre las etiquetas de datos y el parámetro **ax=ax** que corresponde recuadro o figura donde se va a plasmar la grafica. Por ultimo le agregamos un titulo a la grafica.

1. **Matriz de dispersión:** Por medio de esta grafica podremos ver si nuestras variables tienen una tendencia lineal o una interacción de cluster. Es decir como que comporta una variable vs otra. Para eso utilizamos

**Procesamientos de los datos:**

Tenemos que comprender nuestros datos, es decir ¿De que tratan? Por ejemplo:

* ¿Cuántos registros hay?
* ¿Estan todas las final completas o tenemos campos con valores nulos? En caso de que haya demasiado nulos: ¿Queda el resto de información inutil?
* ¿Qué datos son discretos y que datos son continuos? Muchas veces es buenos saber el tipo de dato para saber que trato se le debe de dar o si se debe de realizar algún transformación
* ¿Cuáles pueden ser características importantes? ¿Cuáles podemos descartar?
* ¿Siguen alguna distribución? ¿Hay correlación entre características?
* ¿Cuáles son los Outliers? (Unos pocos datos aislados que difieren del resto y <contaminan> o desvían la distribución)
* ¿Tenemos posibles sesgos de datos?

1. **Necesidad de preprocesamiento:**
   1. Los dato deben de ser procesados en muchos casos:
      1. Por ejemplo, algoritmos tipo árbol se comportan mejor con atributos de tipo nominal.
   2. Problema empirico 🡪 necesidad de probar diferentes transformaciones de datos
   3. Algunas reglas generales:
      1. Los métodos basados en instancias, por ejemplo K-NN, son mas efectivos si los atributos de entrada tienen la misma escala.
      2. Los métodos de regresión pueden funcionar mejor si los atributos de entrada están estandarizados.
2. **Método de transformación:**

Para hacer procesamiento de datos, nos podemos apoyar con el paquete de Python [**sklearn.preprocessing**](https://scikit-learn.org/stable/modules/preprocessing.html)el cual proporciona varias funciones de utilidad comunes y clases de transformadores.

* 1. [**Escalamiento**](https://docs.google.com/document/d/1-Rm8s29CFoC1F8tG0FY0Emmv0ERxSBdd/edit)**:** 
     1. Esta transformación es útil para los algoritmos de optimización utilizados en el núcleo de los algoritmos de aprendizaje automatico como **Gradiente Descendiente**
     2. Tambien es útil para algoritmos que ponderan entradas como Regression y Neural Networks y algoritmos que usan medidas de distancia como **k-Nearest Neighbours**
     3. Puede reescalar sus datos usando la clase [**MinMaxScaler**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html)**.** Quedando en un rango entre 0 y 1.
  2. **Estandarización de los datos:**
     1. Es mas adecuado para técnicas que asumen una distribución gaussiana en las variables de entrada y funcionan mejor con datos reescalados, como LiR (Lineal Regression), LoR y LDA.
     2. Podemos estandarizar datos utilzando la clase [**StandardScaler**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler)
  3. **Normalización de datos:**
     1. Los valores de los datos se pueden escalar en el rango de 0 y 1, es decir, con valores mínimo a 0 y el valor maximo a 1.
     2. Este metodo de preprocesamiento puede ser útil para conjuntos de datos dispersos (muchos ceros) con atributos de escalar variables.
        1. Cuando se utilizan algoritmos que ponderan valores de entrada como NN y algoritmos que usan medidas de distancia como k-NN
  4. **Box-Cox:** Se aplica cuando los datos no tienen una distribución normal se utiliza esta transformación **Box-Cox** para poder hacer que tengan esa distribución normal.
     1. Se aplica cuado los atributos representan un sesgo o inclinación (Gaussiana desplazada)
     2. Box-Cox asume todos los atributos positivos. **Si se tienen valores negativos no se puede aplicar**
     3. Aplica la transformación a los atributos que parecen tener sesgo. No se aplica a las columnas o atributos con distribución normal.
     4. Corrige la no linealidad en la relación (Mejorar la correlacion entre las variables)
     5. Se realiza utilizando la función [**PowerTransformer**](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PowerTransformer.html#sklearn.preprocessing.PowerTransformer)
  5. **Yeo-Johnson:** Funciona igual que **Box-Cox** pero soporta valores que son iguales a 0 y negativos en la columna.

1. **Metodo de remuestreo**

El objetivo es evaluar los algoritmos, para saber que algoritimo se adapta mejor al conjunto de datos. Y para eso debemos de dividir nuestro conjunto de datos para poder realizar una evaluación.

Se tiene cuatro enfoques de división:

* 1. Como dividir un conjunto de datos en subconjuntos por porcentaje para entrenamiento y validación – Donde utilizamos el mayor porcentaje para entrenar el modelo y el porcentaje restante lo dejo por fuera para luego evaluar la precisión del modelo. Por lo general se utiliza un 70/30, esto puede variar de acuerdo con la necesidad.
  2. Como evaluar la robustez del modelo utilizando la validación cruzada, **k-fold,** con y sin repeticiones. – Este corresponde a un modelo que hacer múltiples iteraciones del conjunto de datos, donde en cada iteración divide los datos en entreno y test, y en cada iteración no repite datos. Es decir, que los datos que utilizo para test en la primer iteración no serán los mismo que utilice en la segunda iteración. El modelo se llama **KFold** de la librería [**sklear.model\_selection**](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html#sklearn.model_selection.KFold)**.**

Al utilizar este modelo, el resultado será la media de los accuracy de cada iteración. La idea es que el accuracy sea por encima de 0.8 pero con una desviación por debajo de 0.5

* 1. Como evaluar la robustez del modelo usando una validación cruzada dejando uno fuera (LOOCV)
  2. Division en train y test repetidos aleatoriamente.

**Nota:** El método de evaluación dependiendo del algoritmo, es el **acurracy**

**¿Que técnica usar?**

* + Por lo general, la validación cruzada **K-Fold** es el estándar para evaluar el rendimiento de un algoritmo en datos no etiquetados con K configurado en 3, 5 o 10. Dependiendo que tan grande sea el conjunto de datos.
  + El uso de una división de train y test es buena para la velocidad cuando se usa un algoritmo lento y produce estimaciones de rendimiento con un sesgo mas bajo cuando se usan conjuntos de datos grandes.

1. **Estandarización y escalamiento con Python**
   1. **Re escalamiento con el Metodo MinMax :** Para realizar un escalamiento con el metodo min max, los podemos hacer con la librería **preprocessing** y utilizando la funcion **MinMaxScaler**

**Ej.**

#Realizando escalamiento de datos con el metodo MinMax

from sklearn.preprocessing import MinMaxScaler

df\_int\_2 =df\_int.iloc[:,-5:] #Definimos un DF solo con las ultimas 5 columnas

scaler = MinMaxScaler(feature\_range=(0,1)) #Llamamos la funcion de estandarizacion

Df\_int\_ScalerMinMax = scaler.fit\_transform(df\_int\_2) # Escalamos los datos

Df\_int\_ScalerMinMax = pd.DataFrame(Df\_int\_ScalerMinMax, columns=df\_int\_2.columns)

Df\_int\_ScalerMinMax.describe() #Almacenamos en un DF y vemos el estadistico

En este ejemplo importamos la funcion **MinMaxScaler,** posterior a eso definimos un DF que vamos a utilizar como ejemplo en el cual solo utilizamos las ultimas 5 columnas. Definimos la variable **scaler** en la cual almacenamos el llamdo a la funcion **MinMaxScaler()** y a esta le pasamos el parámetro **feature\_range=(0,1)** lo que significa que el rango en el cual va a realizar la estandarización es entre 0 y 1. Por ultimo utilizamos el **fit\_transform()** donde le pasamos el DF con las variables a estandarizar y el resultado lo almacenamos en una variable que luego lo convertimos en un DF, ya que el resultado de la estandarización sale como array.

* 1. **Estandarizacion con StandardScaler:** Podemos estandarizar los datos utilizando la función **StandardScaler** de la librería **Sklearn.** Donde los valores por cada atributo ahora tiene un valor medio de 0 y una desviación estándar de 1.

**Ej.**

#Realizando Estandarizacion de datos con el metodo MinMax

from sklearn.preprocessing import StandardScaler

df\_int\_2 =df\_int.iloc[:,-5:] #Definimos un DF solo con las ultimas 5 columnas

standard = StandardScaler() #Llamamos la funcion de estandarizacion

Df\_int\_standard = standard.fit\_transform(df\_int\_2) # Estandarizamos los datos

Df\_int\_standard = pd.DataFrame(Df\_int\_standard, columns=df\_int\_2.columns)

Df\_int\_standard.describe() #Almacenamos en un DF y vemos el estadistico

En este ejemplo de estandarización utilizamos el mismo DF del escalamiento, solo que utilizamos la funcion **StandardScaler()** para realizar el escalamiento de las ultimas 5 columnas.

* 1. **Box-Cox y Yeo-Johnson Python:**

En ocaciones los atributos representan el llamado sesgo o inclinación, que es cuando un atributo tiene una distribución similar a la Gaussiana pero esta se encuentra desplazada. Para este tipo de caso, se utiliza la transformación **Box-Cox,** la cual asume que todos los valores son positivos y mayor a 0, que lo que realiza es el sesgo y hace mas Gaussiana la distribución de este atributo (Corrige el sesgo de la distribucion). **Yeo-Johnson** Funciona igual, la única diferencia es qu este si acepta valores negativos e iguales a 0

**Nota;**

En algunos algoritmos, el sesgo da malos resultados y por lo tanto debemos corregirlos.

**Ej Box-Cox:**

#Transformacion con Box-Cox

from sklearn.preprocessing import PowerTransformer

#Extraer caracteristica con sesgo

features = df\_int[["tenure"]]

features = features[features["tenure"]>0]

#Se crea objeto de PowerTransforme

pt = PowerTransformer(method="box-cox", standardize=True)

#Se realiza la transformacion de la columnas

features\_transforms = pt.fit\_transform(features)

#Se almacena el resultado en un DF

features\_transforms\_df = pd.DataFrame(features\_transforms, columns=["tenure"])

print(features\_transforms\_df) #Se muestra el resultado

Para este ejemplo empezamos importando la clase **PowerTransformer** el cual es necesario para utilizar el Box-Cox, posteriormente definimos un DF nuevo que va a contener la columna que vamos a transformar (**Si bien es solo una columna, esta se debe de pasar con doble llave[[ ]] para que quede como un df u no como una serie, ya que Box-Cox no permite series** ), le podemos pasar las columnas que requiramos, para este ejemplo solo se utilizo una. Despues de definir la columnas que vamos a estandariza, filtramos los valores por todo lo que sea mayor a 0 ya que **como lo mencionamos antes, Box-Cox** no permite nulos ni 0s.

Por ultimo, llamamos el objeto **PowerTransformes** al cual le pasamos el parámetro “**method=’box-cox’** ” aca es donde le especificamos el metodo a utilizar, el parámetro “**standardize=True**” de esta manera le indicamos que estandarice los valores. El resultado de la transformación lo almacenamos en un DF, el cual le pasamos la variable con el valor de la transformación y el nombre que va a llevar la columna.

Y graficamos como quedo la distribución de la columna transformada

sns.displot(features\_transforms\_df, x="tenure")

**Nota:** Al momento de crear el DF, el nombre de la columna siempre debe de ir como lista, es decir con llaves []. Adicional, si queremos crear el DF utilizando un diccionario, el resultado de la tranformación debe de ser **unidimensional** de lo contrario no se puede crear por medio de un diccionario (features\_transforms\_df = pd.DataFrame({"tenure": features\_transforms}))

* 1. **Validación cruzada K-fold:**

Como se menciono anteriormente, la validación cruzada es un proceso en la que se realiza **k** particiones o **folds** de la base de datos y con ellos se realiza **k** evaluaciones diferentes.

Y como se menciono anteriormente el resultado será la media de los accuracy de cada iteración. La idea es que el accuracy sea por encima de 0.8 pero con una desviación por debajo de 0.5

**Ej.**

#Se importa libreria del modelo y el evaluador

from sklearn.model\_selection import KFold

from sklearn.model\_selection import cross\_val\_score

from sklearn.linear\_model import LogisticRegression

kfold = KFold(n\_splits=10, random\_state=1)

model = LogisticRegression(solver="lbfgs", max\_iter=1000)

result = cross\_val\_score(model, X, Y, cv=kfold) #Se evalua el resultado

**PENDIENTE TERMINAR LA FUNCION PORQUE NO SE HA DEFINIDO LA VARIABLE Y y X. CONSULTAR MAS SOBRE LOS PARAMETROS DE LOGOSTOCR**

**Fase tratamiento de datos**

1. **Evaluación de métricas:**
   1. **Accuracy:** Es una métrica aplicadas a problemas de **clasificación**  y corresponde al porcentaje de instancias correctamente clasificadas de todas las instancias. Muy útil para la clasificación binaria pero no para clasificación multiclase.
   2. **Kappa o Cohen`s Kappa:** Tambien es una metrica aplicada a problemas de **clasificación** y es útil para usar en problemas que tienen desequilibrio en las clases (Datos desbalanceados).

Lo que hace es penalizar por medio de un método estadístico esa predicción, es decir, que esa clase minoritaria que no esta prediciendo bien, lo que hace es penalizar ese % de acierto, dando un nuevo porcentaje de acierto total. Y la metrica en Python se encuentra como [**cohen\_kappa\_score**](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.cohen_kappa_score.html#sklearn.metrics.cohen_kappa_score)

**Nota:** Si tenemos un conjunto de datos con las clase balanceada no se recomienda utilizar esta metrica, se recomienda utilizar cuando los datos están a partir de un 65/35.

* 1. **Matriz de confusión:** Tambine es aplicada a problemas de **clasificación** y no permite ver cuantas variables se clafican correctamente y cuantas no. Para esto podemos utilizar el metodo **confusión\_matrix** del modulo de **sklearn.metrics**

**Selección de mejores características.**

Corresponde al proceso de seleccionar un subconjunto de características pertinentes (Variables, predictores) para su uso en construcción de modelos.

* ¿**Por qué hacer esto**?
  + Simplificación de modelos
  + Menor tiempo de entrenamiento
  + Reducir el overfitting

1. **Función basada en correlación:** Es donde se mide el grado de relación de dos variables. Determina cunato cambian las caracterisiticas entre si o respecto a la variable objetivo.

Como se mencionó anteriormente, lo ideal es escoger las variables con una correlación superior al 0.75 vs la variable objetivo y que no exista correlación entre las variables predictoras.

1. **Eliminacion recursiva de atributos:** 
   1. **RFE** funciona eliminando recursivamente los atributos y construyendo un modelo sobre los atributos que quedan.
   2. Utiliza la precisión del modelo para identificar que atributos (y combinación de atributos) contribuyen mas a predecir el atributo objetivo.
   3. La librería que se utiliza para este metodo es **RFE** de la librería **SKLEARN.FEATURE\_SELECTION**
2. **Feature importance:**
   1. La importancia de las características se pude estimar a partir de datos mediante la construcción de un modelo.
   2. Algunos métodos como los arboles de decisión tienen un mecanismo incorporado para informar sobre la importancia variable.
   3. Para otros algoritmos, la importancia se puede estimar utilizando un análisis de curva **ROC** realizado para cada atributo.
3. **Decision Trees:** Los arboles de decisión hacen un particionamiento del espacio de entrada mediante una estrategia voraz.
   1. En esta técnica lo que se hace es crear un modelo con la función **DecisionTreeRegressor** de la librería **sklearn.tree** donde se entrena con las variables X y Y
   2. Después de entrenar el modelo podemos utilizar la propiedad **.feature** la cual nos muestra cuales fueron las variables que utilizo para el modelo (las que mayor valor le aporto).
4. **Extra Trees:** Este método también lo que hace es construir multiples árboles y evalúa cuales son las variables que añaden mayor información al modelo. Les asigna un % de participación de importancia a cada variable.
   1. En esta técnica también se entrena un modelo con la función **ExtraTreesClassifier** de la librería **sklearn.ensemble** donde creamos y entrenamos el modelo con las variables de X y Y
   2. Despues de entrenar el modelo utilizadmos la propiedad **.feature\_importances\_** y asi obtenemos el nombre de la variable y su valor de participación en el modelo.
5. **LASSO:** Este es un modelo de tipo regresión lieneal pero que aplica una penalización que se aplica a la característica. Esta también nos indica que variable agrega valor y cual no. **CONSULTAR MAS SONRE ESTA TECNICA**
6. **Reduccion de dimensiones:**

PCA es un procedimiento estadístico que utiliza una transformación ortogonal que convierte un conjunto de variables correlacionadas en un conjunto de variables no correlacionadas. Lo que hace es reducir dimensiones de un conjunto de dato pero asumiendo o intentando asumir la mayor varianza posible que tenia ese conjunto de datos original.

**CONSULTAR MAS**

**Ej.**

# Feature Extraction with PCA

# Libraríes

from sklearn.decomposition import PCA

# PCA con k=3 (es decir 3 componentes)

k=3

pca = PCA(n\_components=k)

fit = pca.fit(X\_cla)

# summarize components

c = pca.components\_

print(f"Explained Variance: {fit.explained\_variance\_ratio\_}")

print(f"components: {c}")

Iniciamos importando la librería que contiene la función **PCA** y luego definimos la cantidad de componentes en los cuales se va a reducir la información. Procedemos a crear y entrenar el modelo, para finalmente ver un resumen de los datos la varianza que componen los 3 componentes utilizando **.** **explained\_variance\_ratio\_**

**Métricas para evaluación de algoritmos – Ej Practico:**

Las métricas que elige para evaluar sus algoritmos de Machine Learning son muy importantes. La elección de métricas influye en cómo se mide y compara el rendimiento de los algoritmos de Machine Learning. Influyen en cómo se valora la importancia de las diferentes características en los resultados y su elección final de que algoritmo elegir.

* **Accuracy:** El cual corresponde al % de acierto y no a una precisiónc– Nos dice cual es el % de acierto que ha tenido el modelo. **El cual es recomendable solo para conjuntos de datos con las clases predictoras balanceadas.** Para datos desbalanceados se puede utilizar **Kappa.** 
  + Una manera de visualizar si el DF tiene la clase desbalanceada podemos agrupar por la variable sobjetivo y utilizamos la función size() para determinar cuantos hay por cada uno.

**Ej.** df\_clas.groupby("class").size()

En este caso al DF\_clas le pasamos la función **groupby()** y a esta le pasamos la columna (variables) por la cual queremos agrupar y por ultimo le pasamos la función **size()** la cual me dice cuantas hay por cada variables

**Nota:** Esta métrica es aplicable a casi cualquier modelo de clasificación como **Random Forest, Support Vector Machines, Gradient Boosting, entre otros.** Siempre y cuando sean datos balanceados y que no sean casos de detección de anomalías donde el 98% es normal y el 2% no. Y para casi cualquier caso se aplica la misma sintaxis que se muestra en el ejemplo practico:

result = cross\_val\_score(model, X\_clas, Y\_clas, cv=kfold, scoring=scoring)

**Ej. Accuracy**

En este ejemplo se mostrara como se puede evaluar un modelo de clasificación utilizando le metodo de validación cruzada (**K-fold**) con la metrica **“Accuracy”**

from sklearn.model\_selection import KFold # Para crear los folds de validación cruzada

from sklearn.model\_selection import cross\_val\_score #Para calcular los scores de validación cruzada

from sklearn.linear\_model import LogisticRegression #Modelo de clasificación a evaluar

# Configuración de la estrategia de validación cruzada

kfold = KFold(n\_splits=10, random\_state=7, shuffle=True)

# Creación del modelo de regresión logística

model = LogisticRegression(solver='lbfgs', max\_iter=1000)

scoring = "accuracy" # Definición de la métrica de evaluación

# Ejecución de la validación cruzada

result = cross\_val\_score(model, X\_clas, Y\_clas, cv=kfold, scoring=scoring)

# Impresión de resultados

print(f'Accuracy {result.mean()\*100:,.2f}% ({result.std()\*100:,.2f})')

Despues de importar las respectivas librerías para realizar la evaluación del modelo procedemos a:

* + Configurar la validación cruzada **kfold** donde le pasamos los siguientes parámetros
    - n n\_splits=10: divide el dataset en 10 partes (folds) para 10 iteraciones de entrenamiento/validación
    - random\_state=7: semilla para reproducibilidad (mismos splits en cada ejecución)
    - shuffle=True: mezcla los datos antes de dividirlos (evita sesgos por ordenamiento)
  + Procedemos a crear el modelo, en este caso uno de regresión logística donde especificamos los siguientes parametros:
    - solver='lbfgs': algoritmo de optimización recomendado para datasets pequeños/medianos
    - max\_iter=1000: aumenta las iteraciones máximas para asegurar convergencia

**Nota:** Si al modelo no se le especifican los parámetros, estos se tomaran los valores por defecto, que en el caso de **solver** es ese mismo (**lbfgs**) y **max\_iter** es 100. En el caso de **solver** cuando se tiene un data set grande se recomienda utilizar “**sag**” o “**saga**” y en el caso **max\_iter** si se deja el valor por defecto puede presentar problemas con data sets complejos, por lo cual es recomendable utilizar 500 o 1000.

* + Definición de la métrica de evaluación que en este caso es **“Accuracy”**
    - "accuracy": que como se menciono antes es la proporción o % de predicciones correctas (TP+TN)/(TP+TN+FP+FN)
  + Se realiza la ejecución de la validación cruzada con al función **cross\_val\_score()**:
    - model: modelo a evaluar – Definido previamente (modelo de regresion)
    - X\_clas: variables predictoras (features)
    - Y\_clas: variable objetivo (target)
    - cv=kfold: estrategia de validación (10-fold) – El cual corresponde al que se parametrizo inicialmente
    - scoring=scoring: métrica a utilizar (accuracy)
  + Y por ultimo mostramos el resultado de la metrica:
    - result.mean(): accuracy promedio de los 10 folds (convertido a porcentaje)
    - result.std(): desviación estándar de los accuracy (variabilidad entre folds)

**Nota:** Para la interpretación del resultado, un buen modelo tendrá un alto **Accuracy** promedio (cercano a 100%) y baja **desviación estándar** (Resultados consistentes entres folds)

**Nota del ejemplo:** En el código no se aplica división del conjunto de datos como por ejemplo X\_train y X\_test, y tampoco se aplica un entrenamiento del modelo ni una prediccion, porque el modelo **cross\_val\_score** y **Kfold** lo hacen automáticamente en cada iteración de fold.

Tambien debemos de tener presente que podemos aplicar la metrica **Accuracy** de manera directa, es decir, utilizando la función **accuracy\_score()** de la librería **Skalearn.metrics.** En este caso si debemos de aplicar una separación de datos de manera manual, creación y predicción del modelo.

Igual que el cross\_val\_score el **accuracy\_score** también se aplica a los modelos de clasificación mencionados anteriormente (SVM, Random Forest, etc)

**Ej aplicación Accuracy directo:**

#Aplicacion de Accuracy sin validacion cruzada (Kfold)

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score

# 1. Separación manual de datos (train/test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_clas, Y\_clas, test\_size=0.2, random\_state=7)

# 2. Creación y entrenamiento del modelo

model = LogisticRegression(solver='lbfgs', max\_iter=1000)

model.fit(X\_train, y\_train) # Entrenamiento explícito

# 3. Predicción y cálculo de accuracy

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)# Aplica para cualquier modelo de clasif

print(f'Accuracy: {accuracy\*100:.2f}%')

* **Kappa:** La metrica **Kappa o Cohen’s Kappa** es como una precisión de clasificación (Accuracy) pero ajustado **cuando se tienen datos desbalanceados**. Es decir que es una medida mas útil para usar en problemas que tiene un desequilibrio en las clases (por ejemplo, una división de 70% al 30% para la case 0 y 1, respectivamente y puede alcanzar el 70% de precisión.)

**Ej Metrica Kappa:**

# Cross Validation Classification Confusion Matrix Kappa

# Importación de librerías necesarias

from sklearn.model\_selection import train\_test\_split #Dividir dataset train/test

from sklearn.metrics import cohen\_kappa\_score #Calcular la métrica Kappa de Cohen

from sklearn.linear\_model import LogisticRegression #Modeloa evaluar (Clasific..)

# Configuración de parámetros para la división de datos

test\_size = 0.33 # 33% de los datos para test, 67% para entrenamiento

seed = 7 # Semilla para reproducibilidad (misma división cada ejecución)

# División del dataset en conjuntos de entrenamiento y prueba

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(

X\_clas,

Y\_clas,

test\_size=test\_size,

random\_state=seed) # Fija la semilla para resultados reproducibles

# Creación del modelo de regresión logística

model = LogisticRegression(solver="lbfgs", max\_iter=1000)

# Entrenamiento del modelo con los datos de entrenamiento

model.fit(X\_train, Y\_train)

# Generación de predicciones con los datos de prueba

predict = model.predict(X\_test)

# Cálculo del Kappa Score comparando predicciones vs valores reales

kappa\_score = cohen\_kappa\_score(Y\_test, predict)

# Visualización del resultado formateado como porcentaje con 2 decimales

print(f'Kappa Score: {kappa\_score\*100:,.02f}%')

En este ejemplo estamos evaluando un modelo de clasificación **LosgisticRegression** por medio de la meteica **Kappa** (**Cohen\_kappa\_score**)

* + Empezamos importando las librerías requeridas
  + Creamos 2 variables con parámetros que le vamos a pasar a la función que no va a dividir el DF en traint y test
  + Utilizamos la función **train\_test\_split** para separar las variables independientes (X) y la variable dependiente (Y) en train y tetst.
    - Le pasamos las variable X y Y
    - Le indicamos con le parámetro **test\_size** que porcentaje queremos para trein y que porcentaje para test, en este caso 67% y 33% respectivamente, corresponde al valor que le asignamos a la variable **test\_size**
    - En el parámetro **random\_state** le indicamos cual será la semilla para reproducibilidad, en este caso será 7 y corresponde al valor que le asignamos a la variable **seed**
  + Creamos el modelo de regresión logística donde le pasamos los siguientes parámetros:
    - **solver**="lbfgs": Algoritmo de optimización recomendado para datasets pequeños/medianos
    - **max\_iter**=1000: Aumenta iteraciones máximas para asegurar convergencia

**Nota:** Para estos parámetros aplica lo mismo que se explica en el ejemplo del modelo para la metrica **Accuracy**

* + Se entrena el modelo con los datos de entrenamiento (X\_train y Y\_train)
  + Se genera predicciones con los datos de prueba (X\_test)
  + Cálculo del **Kappa Score** comparando predicciones vs valores reales (Y\_test)
  + Por ultimo visualizamos el resultado de la metrica **Kappa**
    - **Interpretacion:** Si se optiene un Score Kappa por encima de 80% es un resultado excelente y si el resultado esta entre 61% y 80% es un buen resultado. Si es menos a 60 se recomienda revisar o **probar otros modelos de clasificación** y volver a evaluar.

**Nota:** La métrica de Kappa también se aplicar con el modelo de **cross\_val\_score** pero no directamente como se hace en el caso del **Accuracy** que se le pasa al parámetro **scoring=.** Para usar **Kappa** con **cross\_val\_score**, podemoss definirlo como una métrica personalizada utilizando la función **cohen\_kappa\_score** de **sklearn.metrics** y pasarla a **cross\_val\_score** mediante **make\_scorer** también de **sklearn.metrics.**# Crea un scorer personalizado para Cohen's Kappa

kappa\_scorer = make\_scorer(cohen\_kappa\_score)

* **Metrica bajo la curva ROC**

Es una metrica que tambien se le conoce como **AUC** y es una métrica muy importante para la clasificación binaria. El **AUC** representa la capacidad de un modelo para descriminar entre clases positivas y negativas. Un área de 1.0 representa un modelo que hizo todas las predicciones perfectas. Un área de 0.5 representa un modelo tan bueno como aleatorio.

**Ej. Metica ROC (AUC)**

La aplicación de esta métrica, es similar a las explicadas anteriormente (**Accuracy** y  **Kappa**) en este ejemplo vamos aplicar la métrica por medio de la validación cruzada utilizando **Kfold** y **cross\_val\_score.**

# Cross Validation Classification ROC AUC

# Importación de librerías necesarias

from sklearn.model\_selection import KFold # Para crear los folds de validación cruzada

from sklearn.model\_selection import cross\_val\_score # Para calcular los scores de validación cruzada

from sklearn.linear\_model import LogisticRegression # Modelo de clasificación a evaluar

# Configuración de la estrategia de validación cruzada

# Recuerda que la nueva versión obliga poner shuffle

kfold = KFold(n\_splits=10, random\_state=7, shuffle=True)

# Creación del modelo de regresión logística

modelo =LogisticRegression(solver="lbfgs", max\_iter=1000)

# Ejecución de la validación cruzada

roc\_score = cross\_val\_score(modelo, X\_clas,Y\_clas, cv=kfold, scoring="roc\_auc")

#Se muestra el resultado de la metrico

print(f'La metrica ROC: {roc\_score.mean()} ({roc\_score.std()})')

Como en los ejemplos anteriores, empezamos importando las librerías requeridas para crear y evaluar el modelo. Los pasos que se aplican en la aplicación de esta metrica son los mismo que se vienen aplicando en la validación cruzada de los pasos anteriores

* + Configuramos la validación cruzada **Kfold**  y le pasamos los parámetros requeridos, **n\_split (**numero en el que se va a divir el df**), random\_state (**semilla de reproducibilidad**), shuffle(**combinación de datos que aplica antes de dividir el df**).**
  + Cramos el modelo de clasificación
  + Ejecutamos el modelo con la metrica a evaluar ROC, que se debe de pasar en el **Scoring=** como “roc\_auc”. Por ultimo mostramos el resultados

**Interpretación:** Como en las meteicas anteriores, un buen resultado es cuando esta mas cera a 1.0 y cuando esta por debajo de 0.6 quiere decir que el modelo no esta clasificando bien.

**Si se requiere aplicar la metrica a modelos de multiclase, se le debe de realizar una adaptación, ya que su foco principal es evaluar la clasificación binaria.**