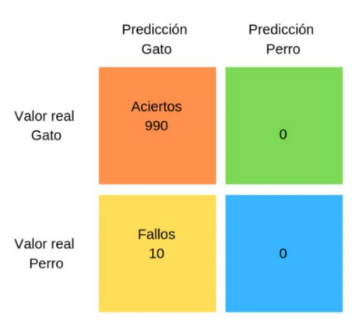
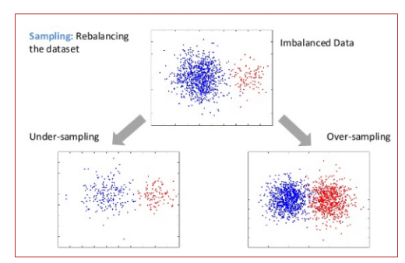
**Clases Desbalanceadas / Features Selection**

Es común encontrar problemas de clasificación donde alguna de las clases sea minoritaria, porque disponemos de pocos casos. Esto provoca un **desbalanceo de clases** que los modelos no siempre detectan. Los modelos buscan predecir las clases mayoritarias pero muchas veces lo que nos interesa es la clase minoritaria. Ejemplos típicos de este tipo de problemas son el diagnóstico de enfermedades o la detección de operaciones fraudulentas. Por ejemplo, entrenamos un modelo de clasificación con 990 imágenes de gatos y 10 imágenes de perros. El modelo acertaría el 99% de los casos clasificando todo como gatos:



El desbalanceo de clases se puede atacar de tres maneras:

1. Haciendo un resampling, aumentando los casos de la clase minoritaria (**oversampling**).
2. Haciendo un resampling, descartando casos de la clase mayoritaria (**undersampling**).
3. Equilibrando la clase minoritaria, dándole un peso mayor (**Class Weighting**).

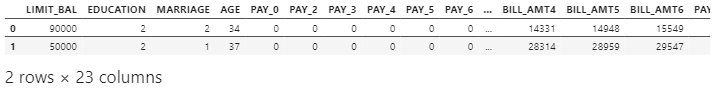


Ejemplo **en Python:**

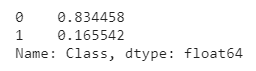
Vamos a usar un DataSet de incumplimientos de Tarjetas de Crédito, siendo las observaciones clientes de tarjetas de crédito de Taiwan; y la clase si la persona está en incumplimiento (1) o no (0).

df\_original = pd.read\_excel(‘../Data/default\_credit\_card.xls’)

df\_original.head(2)



print(pd.value\_counts(df[‘Class’], sort = True, normalize = True))



# Vamos a aplicar un modelo de regresión logística y medir su performance:

X = df.drop([‘Class’], axis = 1)

y = df[‘Class’]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3, random\_state = 123)

scaler = StandardScaler()

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

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

def run\_model(X\_train, y\_train):

clf\_base = LogisticRegression(penalty = ‘none’, random\_state = 123, max\_iter = 500)

clf\_base.fit(X\_train, y\_train)

return clf\_base

def mostrar\_resultados(y\_test, pred\_y):

conf\_matrix = confusión\_matrix(y\_test, pred\_y)

plt.figure(figsize=(5,3))

sns.heatmap(conf\_matrix, annot = True, fmt = ‘d’)

plt.title(‘Confusion Matrix’)

plt.ylabel(‘True class’)

plt.xlabel(‘Predicted class’)

plt.show()

print(‘METRICS’)

print(classification\_report(y\_test, pred\_y))

# Ahora vamos a aplicar el modelo sobre el dataset original. Vamos a ver que con un accuracy de 0.85, el modelo es muy bueno prediciendo la clase mayoritaria pero es horrible con la minoritaria:

model\_original = run\_model(X\_train\_sc, y\_train)

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

mostrar\_resultados(y\_test, y\_pred)



Python tiene un módulo llamado **imbalanced-learn** que ayuda a balancear los datasets. Con el método **RandomUnderSampler** es posible tomar observaciones de la clase mayoritaria al azar, con o sin reposición, para descartarlos y hacer un **undersampling**. Cuenta con un hiperparámetro **sampling\_strategy** que puede tomar como valor “minority” o “majority” (hace resampling para balancear las clases); o entre 0 y 1 (ratio entre las clases mayoritaria y minoritaria).

**En Python:**

from imblearn.under\_sampling import RandomUnderSampler

undersampler = RandomUnderSampler(sampling\_strategy = ‘majority’, random\_state = 123) # iguala las clases.

X\_train\_us, y\_train\_us = undersampler.fit\_resample(X\_train, y\_train)

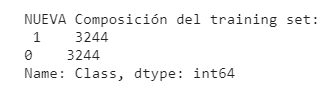
scaler = StandardScaler()

X\_train\_us\_sc = scaler.fit\_transform(X\_train\_us)

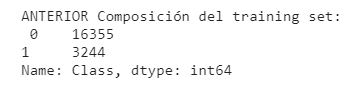
X\_test\_us\_sc = scaler.transform(X\_test)

Al usar sampling\_strategy = ‘majority’, lo que logramos es que el dataset de entrenamiento tenga la misma cantidad de observaciones por clase. Fue reduciendo la cantidad de observaciones en la clase mayoritaria hasta obtener una cantidad igual al total original de la clase minoritaria.

print(‘NUEVA Composición del training set:/n’, y\_train\_us.value\_counts())



print(‘ANTERIOR composición del training set:/n’, y\_train.value\_counts)



Si aplicamos undersampling sobre el dataset, el accuracy baja de 0.85 a 0.69. Esto es porque al reducir fuertemente la cantidad de casos de la clase mayoritaria lleva al modelo a equivocarse más en esta. Otra métrica que baja acompañada del accuracy es la **recall**.

En la clase minoritaria, en cambio, aumenta la sensitivity, ya que se predicen mejor estos casos que sin balancear las clases. Precision baja drásticamente de 0.69 a 0.3.

**En Python:**

model\_us = run\_model(X\_train\_us, y\_train\_us)

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

mostrar\_resultados(y\_test, y\_pred)



Ahora vamos a probar qué pasa si aplicamos el método contrario: **Oversampling**. Vamos a generar un set de datos de entrenamiento con clases balanceadas, a partir de aumentar los casos de la clase minoritaria. Podemos lograr esto aplicando el método **RandomOverSampler,** que crea observaciones de la clase minoritaria con reposición. Este método cuenta también con el hiperparámetro **sampling\_strategy**.

**En Python:**

from imblearn.over\_sampling import RandomOverSampler

# iguala las clases:

oversampler = RandomOverSampler(sampling\_strategy = ‘minority’, random\_state = 123)

X\_train\_os, y\_train\_os = oversampler.fit\_resample(X\_train, y\_train)

Scaler = StandardScaler()

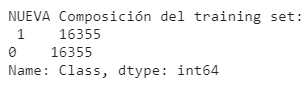
X\_train\_os\_sc = scaler.fit\_transform(X\_train\_os)

X\_test\_os\_sc = scaler.transform(X\_test)

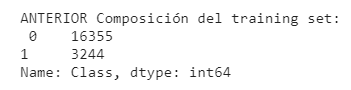
El nuevo train set ahora tiene la misma cantidad de observaciones por clase, pero el camino que usó para lograrlo ue agregar casos repetidos de la clase minoritaria.

**En Python:**

print(‘NUEVA Composición del training set :/n’, y\_train\_os.value\_counts())



print(‘ANTERIOR Composición del training set :/n’, y\_train.value\_counts())



Si estudiamos la performance con este nuevo dataset, vemos que es prácticamente igual a la que conseguimos aplicando undersampling. Otro camino que tenemos es jugar con los valores entre 0 y 1 del hiperparámetro sampling\_strategy para mejorar la performance.

**En Python:**

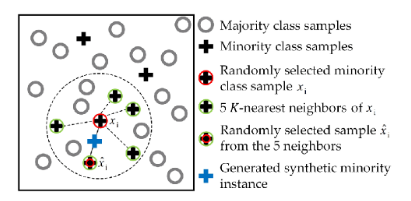
model\_os = run\_model(X\_train\_os\_sc, y\_train\_os)

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

mostrar\_resultados(y\_test, y\_pred)



Con el algoritmo **SMOTE** (**S**ynthetic **M**inority **O**versample **Te**chnique) es posible realizar oversampling generando muestras simuladas de la clase minoritaria. Considera de a pares casos de la clase minoritaria y genera aleatoriamente un punto que se encuentre en el segmento que los une:



Pero SMOTE funciona **solamente con variables continuas.**

Existe también el método **SMOTENC**, que funciona con **variables continuas y con variables categóricas**. En este caso lo que hay que hacer es informarle al modelo las variables categóricas como una lista con la posición de cada feature en el dataset, en el parámetro **categorical\_features**.

**En Python:**

from imblearn.over\_sampling import SMOTENC

smote = SMOTENC(categorical\_features[1, 2, 4, 5, 6, 7, 8, 9], sampling\_strategy = ‘minority’, random\_state = 123)

X\_train\_sm, y\_train\_sm = smote.fit\_resample(X\_train, y\_train)

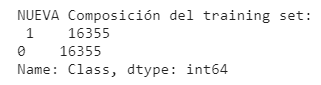
Scaler = StandardScaler()

X\_train\_sm\_sc = scaler.fit\_transform(X\_train\_sm)

X\_test\_sm\_sc = scaler.transform(X\_test)

# El nuevo set de entrenamiento va a tener la misma cantidad de observaciones por clase, pero en este caso lo que hizo es igualar la cantidad de la clase mayoritaria a partir de **datos inventados de la clase minoritaria**.

print(‘NUEVA Composición del training set :/n’, y\_train\_sm.value\_counts())



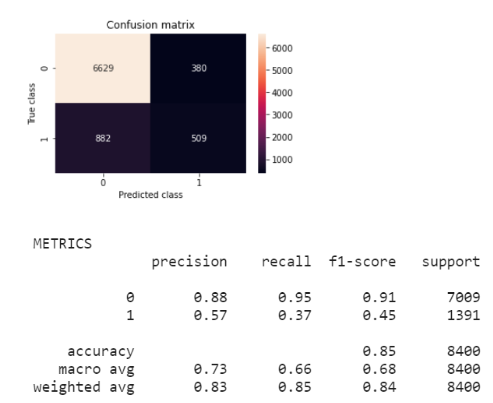
A nivel performance, el resultado es prácticamente igual a los casos previos.

**En Python:**

model\_sm = run\_model(X\_train\_sm\_sc, y\_train\_sm)

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

mostrar\_resultados(y\_test, y\_pred)



Otro camino posible más es **penalizar** o **darle un peso a cada label** de la clase. Esto sirve para imponerle un costo al modelo cuando falla al clasificar la clase minoritaria, balanceando las clases de esta manera. En la regresión logística, se puede usar el hiperparámetro **class\_weight**, donde en un diccionario se pueden definir los pesos por label. Si usamos **class\_weight = balanced**, el algoritmo va a asignar pesos a la inversa de la frecuencia de cada label. En estos casos, se balancean las clases sin hacer resample.

**En Python:**

model = LogisticRegression(class\_weight = ‘balanced’, penalty = ‘none’, random\_state = 123, max\_iter = 500)

scaler = StandardScaler()

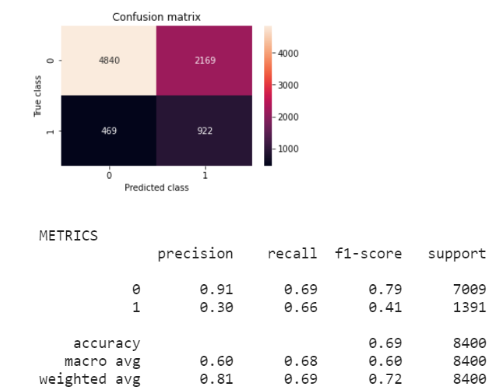
X\_train\_sc = scaler.fit\_transform(X\_train)

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

model.fit(X\_train\_sc, y\_train)

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

mostrar\_resultados(y\_test,y\_pred)

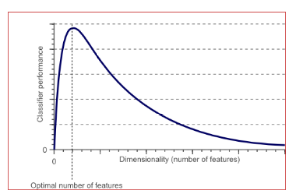


En resumen: Generalmente, los modelos de Machine Learning suelen estar sesgados hacia la clase mayoritaria, cuando en general lo que necesitamos es predecir correctamente la clase minoritaria. Con todas las metodologías mostradas, lo que hacemos es apuntar a mejorar el recall. Hay que probarlas e ir modificando los parámetros para llegar a la mejor performance:

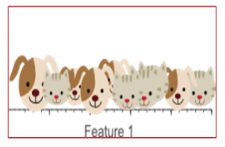


**La Maldición de la Dimensionalidad**:

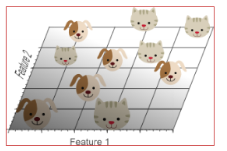
No necesariamente más features implica mejores modelos. Se define un **DataSet de** **alta dimensionalidad** como aquél que tiene **cien o más features**.Ir agregando features va mejorando la performance del DataSet hasta cierto punto, a partir del cual el clasificador empieza a empeorar. A esto Richard Bellman lo llama la **maldición de la dimensionalidad:**

****

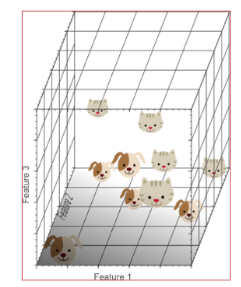
IE: Tenemos 10 imágenes de gatos y perros usando el modelo de color RGB (colores rojo, verde y azul). Si tenemos que construir un clasificador en base a estas imágenes y usamos solo el feature “rojo”, el clasificador no va a funcionar bien.



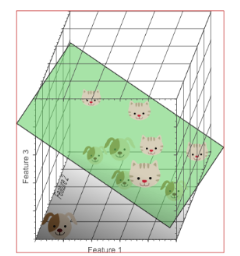
Si agregamos la feature “verde”, entonces ya podemos separar un poco mejor las observaciones:



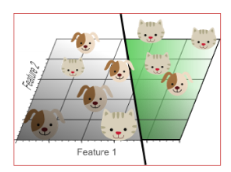
Agreguemos “azul” ahora y se logra una mayor dispersión aún:



A partir de la combinación lineal de rojo, verde y azul, podemos encontrar un plano que separe los gatos de los perros:



Hasta ahora, agregando features, estamos mejorando la capacidad del modelo. A medida que **vamos agregando features**, el **espacio de los predictores** **aumenta**, y los datos se van volviendo **más dispersos** (**sparse**). Esto va volviendo cada vez **más fácil encontrar un hiperplano que separe las clases**, ya que va disminuyendo la probabilidad de encontrar un dato mal clasificado hasta volverse infinitesimalmente pequeña. **Pero**, corremos el **riesgo de overfitear**. En el ejemplo recién expuesto, un modelo de 2 dimensiones es más simple y generaliza mejor que uno de tres:



Otro problema que acarrea agregar variables, es que al crecer la dimensionalidad del problema, cada vez más casos se ubican en los extremos del espacio, es decir, **aumentan los outliers**: Al **aumentar las variables**, **aumenta la probabilidad** de que **al menos en una dimensión** la variable tome un **valor extremo**. Si generamos un espacio demasiado grande, podemos perder la estructura que buscamos encontrar.

IE: Tenemos una variable distribuida uniformemente en un **hipercubo de d dimensiones**. Vamos a ver cómo crecen los outliers. Si alguna de las coordenadas de estos puntos se encuentra en el percentil 1 o 99 de alguna de sus coordenadas, entonces consideraremos que se trata de un outlier. Con dos dimensiones, creamos un marco cuadrado y encontramos unos pocos.

**En Python:**

N = 500

# Generamos 500 puntos en un espacio de dos dimensiones, distribuidos uniformemente.

X = np.random.uniform(size=(N,2))

plt.figure(figsize=(5,3))

plt.plot(X,[:,0],X[:,1], ‘o’, ms=2)

x = [0.0,1.0]

p = 0.01 # Si alguna de las coordenadas está en el percentil 1 o 99 del eje, lo consideramos un outlier.

plt.fill\_between(x, 0, p, Alpha=0.2, color = ‘k’)

plt.fill\_between(x, 1-p, 1, Alpha=0.2, color = ‘k’)

plt.fill\_betweenx(x, 0, p, Alpha=0.2, color = ‘k’)

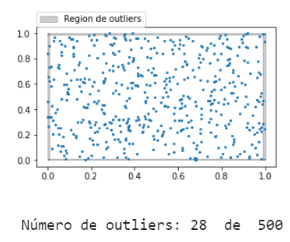
plt.fill\_betweenx(x, 1-p, 1, Alpha=0.2, color = ‘k’, label = ‘Region de outliers’)

plt.legend(loc=(0,1))

plt.show()

n\_outliers = np.sum(np.sum((X<p)|(X>(1-p)), axis = 1)

print(‘Número de outliers:’, n\_outliers, ‘ de ‘,N)



**Si aumentamos la dimensionalidad del dataset**, **para un número fijo de casos**, **crece la proporción de outliers**.

N = 1000

P = 0.01

Ds = np.arange(1,200)

p\_outliers = []

for d in Ds:

X = np.random\_uniform(size = (N,d))

P\_outliers.append(np.mean(np.any((X<p) | (X > (1 - p)), axis = 1)))

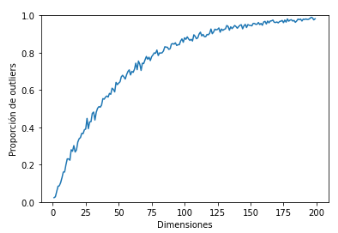
plt.figure(figsize=(6,4))

plt.plot(Ds, p\_outliers)

plt.ylim([0,1])

plt.ylabel(‘Proporción de outliers’)

plt.xlabel(‘Dimensiones’)



**Feature Selection**:Una buena solución cuando tenemos **alta dimensionalidad** es **reducir el número de features**. Si fuéramos probando todas las combinaciones posibles (**fuerza bruta**)para encontrar el mejor conjunto, eso sería computacionalmente inviable. Con **Feature Selection**, lo que hacemos es buscar **identificar y seleccionar** las **variables más relevantes** para el **entrenamiento de modelos**. O, mirándolo desde otro ángulo, sacar aquellas variables que no aportan información o son redundantes.

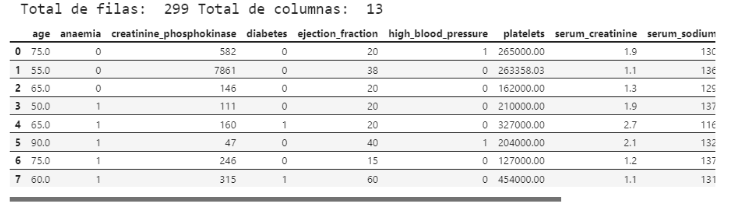
Vamos a trabajar con un DataSet de predicciones de supervivencia de pacientes que ya han tenido ataques cardíacos anteriormente. El mismo cuenta con variables categóricas que indican factores de riesgo (1) (anemia, diabetes, high\_blood\_pressure, smoking); sex (0: mujer, 1: hombre); age; medidas de componentes en la sangre (creatinine\_phosphokinase, platelets, serum\_creatinine, serum\_sodium, una medida de cómo trabaja el corazón (ejection\_fraction) y los días de seguimiento (time). Por último, la variable objetivo es DEATH\_EVENT(0: Sobrevivió; 1: Murió).

**En Python:**

df\_heart = pd.read\_csv(‘../Data/heart\_failure\_clinical\_records\_dataset.csv’)

print(‘Total de filas: ’, df\_heart.shape[0],’total de columnas: ’, df\_heart.shape[1])

df\_heart.head(8)



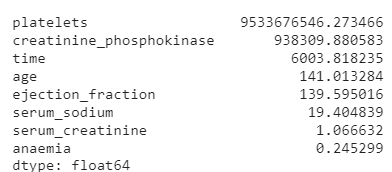
**Filter Methods:** Lo que hacen es rankear a las features en función de su **importancia**. En general, se define un **umbral para los scores**, por **debajo del cual** las **variables** se consideran **poco relevantes** y se filtran. Los **métodos varían** según **cómo se mide esta relevancia**. Una vez que encontramos las mejores features, podemos generar los modelos.

Con la **clase VarianceThreshold**, podemos remover las features que no aportan información al DataSet a partir de su varianza: Se **filtran todas aquellas features** que tengan una **varianza menor** a un **umbral (threshold) definido**. Vamos a aplicar este concepto en el DataSet calculando manualmente la varianza para comprobar el resultado:

**En Python:**

pd.options.display.float\_format = ‘{:.6f}’.format

df\_heart.apply(np.var).sort\_values(ascending=False)[:,8]



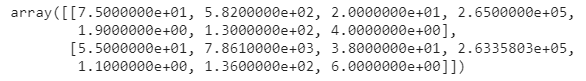
# Ahora vamos a ajustar el DataSet a la Clase, con el **hiperparámetro threshold** = 0.5. Con **fit\_transform** obtenemos un array cuyas columnas son las features seleccionadas.

from sklearn.feature\_selection import VarianceThreshold

vt = VarianceThreshold(threshold = 0.5)

df\_heart\_vt = vt.fit\_transform(df\_heart)

df\_heart\_vt[:2]



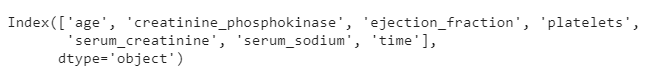
# Con el método **get\_support()** conseguimos una máscara con las features seleccionadas.

vt.get\_support()



# Ahora vamos a aplicarlo al dataSet para conocer cuáles son las features:

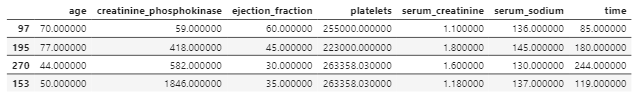
df\_heart.columns[vt.get\_support()]



# Para terminar, podemos construir un DataSet a partir de las features seleccionadas:

df\_heart\_reduced\_vt = pd.DataFrame(df\_heart\_vt, columns = df\_heart.columns[vt.get\_support()])

df\_heart\_reduced\_vt.sample (4)



Con los métodos de **Univariate feature selection** es posible seleccionar y rankear features a partir de **tests estadísticos univariados**. Lo que hacen es analizar las propiedades estadísticas de las features y cuán fuerte es su relación con la variable target. Son **univariados** porque **no contemplan la relación entre features**. Los más usados son:

* Coeficiente de Correlación de Pearson, que se llama con la función **f\_regression()**
* ANOVA: **f\_classif()**
* Chi-cuadrado: **chi2()**
* Mutual Information: **mutual\_info\_classif()** y **mutual\_info\_regression()**

Dos métodos de scikit-learn son **SelectKBest** y **SelectPercentile**.

La Clase **SelectKBest** es una suite de estadísticos para seleccionar las features más significativas.Necesita de dos hiperparámetros:

1. **score\_func**: función que devuelve algún score entre X (features) e y (variable objetivo). Para **clasificación** son **f\_classif**, **mutual\_info\_classif** y **chi2**. Para **regresión** son **f\_regression** y **mutual\_info\_regression**.
2. **k**: cantidad de mejores features a ser seleccionadas.

**En Python:**

Vamos a aplicar Chi-Cuadrado para seleccionar las 10 mejores features del dataset:

X = df\_heart.drop(‘DEATH\_EVENT’, axis = 1)

y = df\_heart[‘DEATH\_EVENT’]

from sklearn.feature\_selection import SelectKBest, chi2

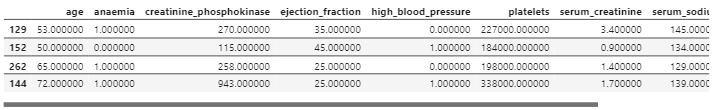
best\_features\_k = SelectKBest(score\_func = chi2, k = 10)

# Con el método fit\_transform selecciona las features más relevantes y las devuelve como un array.

fit\_k = best\_features\_k.fit\_transform(X,y)

df\_heart\_reduced = pd.DataFrame(fit\_k, columns = X.columns[best\_features\_k.get\_support()])

df\_heart\_reduced.sample(4)



# Podemos ver el score de cada Feature:

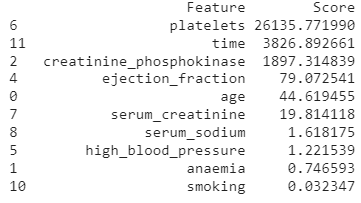
dfscores = pd.DataFrame(bestfeatures\_k.scores\_)

dfcolumns = pd.DataFrame(X.columns)

scores = pd.concat([dfcolumns, dfscores], axis = 1)

scores.columns = [‘Feature’, ’Score’]

print(scores.nlargest(10, ‘Score’))



Con la clase **SelectPercentile** lo que se hace es seleccionar las features en función de los percentiles de los Scores, por ejemplo, las que estén en el mejor 20%. Usa dos hiperparámetros:

1. **score\_func**: una función que devuelva algún score entre X (features) e y (variable objetivo).
2. **percentile**: porcentaje de las features a considerar.

**En Python:**

from sklearn.feature\_selection import SelectPercentile, chi2

best\_features\_p = SelectPercentile(chi2, percentile = 20)

# El método fit\_transform aplica el estadístico y selecciona sólo el % definido más relevante de features:

fit\_p = best\_features\_p.fit\_transform(X,y)

# Con el método **get\_support()** obtenemos una máscara con las features seleccionadas.

# Con el método get\_support() podemos obtener una máscara con las features seleccionadas.

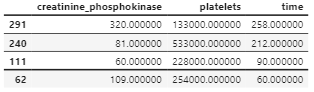
bestfeatures\_p.get\_support()



# Ahora podemos construir un DataFrame que sólo contenga las features seleccionadas:

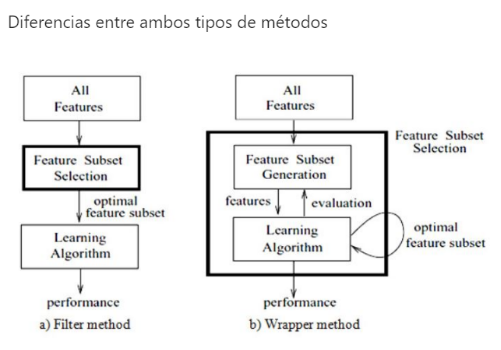
df\_heart\_reduced\_p = pd.DataFrame(fit\_p, columns = X.columns[bestfeatures\_p.get\_\_support()])

df\_heart\_reduced\_p.sample(4)



# Con este método se seleccionaron sólo 3 variables.

**Wrapper Methods:** Estos métodos **aplican un algoritmo sobre un subconjunto de variables** y **en función de la performance** obtenida, va **agregando o removiendo features**. Este algoritmo **se vuelve a aplicar recursivamente** hasta llegar a un criterio de parada. Son métodos más costosos computacionalmente, pero son más precisos que los filter methods. Se considera a la selección del conjunto de features como un **problema de búsqueda**, dondese aplican distintas estrategias.



**Eliminación Recursiva de Features** (**R**ecursive **F**eature **E**limination). Se entrena un **estimador** **sobre todas las features** y se **calcula la importancia de cada una**. **Elimina la menos importante** y **vuelve a entrenar con las restantes**. **Repite** el proceso recursivamente hasta llegar a un **número de features definido previamente**.

Este método cuenta con los siguientes argumentos:

* **estimator:** Estimador usado para entrenar y evaluar. Cualquier algoritmo supervisado que devuelva la importancia de cada feature.
* **n\_features\_to\_select**: Cantidad de features final deseada.
* **steps:** features que se eliminan por iteración.

**En Python:** Vamos a trabajar sobre el DataSet de vinos que tiene como features sus propiedades químicas y como variable objetivo la clasificación en 3 tipos de vino (0,1,2).

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_sc = scaler.fit\_transform(X)

# Vamos a usar como clasificado un árbol de decisión, informando 6 como cantidad de features objetivo (n\_features\_to\_select = 6) y eliminando de a uno (step = 1).

from sklearn.tree import DecisionTreeClassifier

from sklearn.feature\_selection import RFE

estimator = DecisionTreeClassifier()

rfe = RFE(estimator, n\_features\_to\_select = 6, step = 1)

rfe.fit(X\_sc, y)

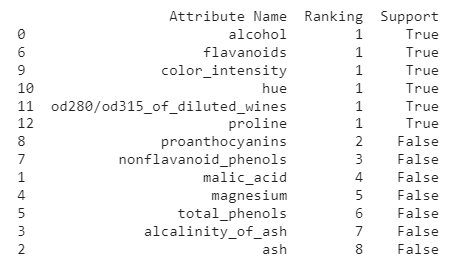
scores = pd.DataFrame()

scores[‘Attribute Name’] = X.columns

scores[‘Ranking’] = rfe.ranking\_

scores[‘Support’] = rfe.support\_

print(scores.sort\_values(‘Ranking’))



**Eliminación Recursiva de Features con Cross Validation** (**R**ecursive **F**eature **E**limination **C**ross **V**alidation)

**Similar a RFE**, pero **determina la cantidad de features** a seleccionar usando **Cross Validation.** Sus argumentos son similares, pero en lugar de informarle la cantidad de variables final deseada, debemos informarle cómo queremos que haga el Cross Validation.

* **estimator:** Estimador usado para entrenar y evaluar. Cualquier algoritmo supervisado que devuelva la importancia de cada feature.
* **CV**:Determina el método de validación cruzada.
* **steps**: Features que se eliminan por iteración.

**En Python:**

from sklearn.tree import decisionTreeClassifier

from sklearn.feature\_selection import RFECV

from sklearn.model\_selection import StratifiedKFold

estimator = DecisionTreeClassifier()

kf = StratifiedKFold(n\_splits = 5, shuffle = True)

rfecv = RFECV(estimator, cv = kf, step = 1)

rfecv.fit(X\_sc, y)

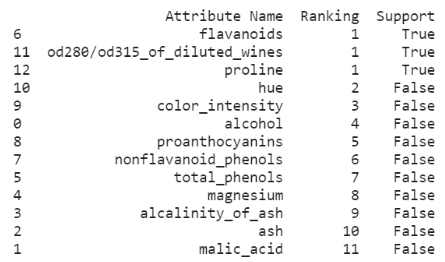
scores = pd.DataFrame()

scores[‘Attribute Name’] = X.columns

scores [‘Ranking’] = rfecv.ranking\_

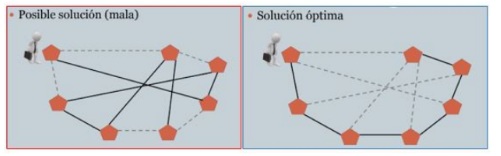
scores[‘Support’] = rfecv.support\_

print(scores.sort\_values(‘Ranking’))



**Algoritmos Genéticos:** Creados porJohn Holland en 1970. Setrata de algoritmos inspirados en la evolución biológica y su base genético-molecular. Se trata de un **método de búsqueda y optimización**. El espacio de búsqueda de un problema determinado está formado por todas las posibles soluciones a dicho problema.

Ejemplo: Problema del viajante: Travelling Salesman Program (TSP): Dada una lista de ciudades y las distancias entre cada par de ellas, cuál es la ruta más corta posible que visita cada ciudad exactamente una vezy al finalizar regresa a la ciudad origen?

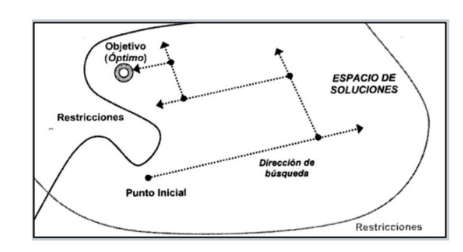


El espacio de búsqueda son todos los posibles caminos sobre el grafo que pasen una sola vez por ciudad. La cantidad de caminos posibles es el factorial de 6 (6! = 720). Cada uno de estos caminos es una solución dentro del espacio total de soluciones. Cada camino se puede representar como un vector de 6 elementos, donde cada uno representa una ciudad.

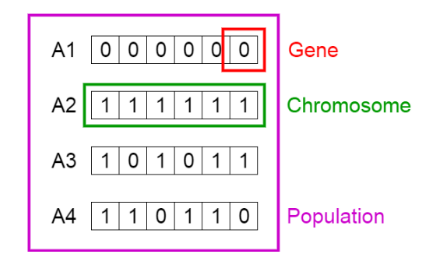
(x1,x2,x3,x4,x5,x6) donde xi es una ciudad entre 1 y 6 y xi != xj.

(1,2,3,4,5,6) es el camino óptimo.

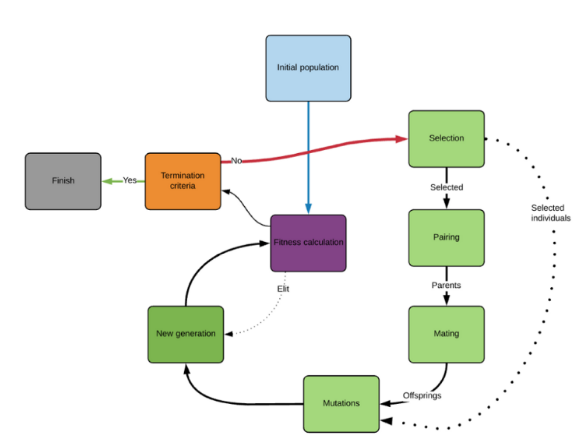
Los **algoritmos genéticos** trabajan sobre una **población de individuos** donde **cada individuo es una solución** dentro del **espacio de soluciones**. Evolucionan la población (se mueven de un conjunto de individuos a otro) aplicando acciones semejantes a las que actúan en la evolución biológica. “El proceso de selección natural”. **Cada** **iteración o** **generación** encuentra **individuos “más aptos”**. El algoritmo termina cuando encuentra la mejor solución, **no siempre los óptimos**, o después de un número de generaciones dado.



Cada individuo de la población está representado por un **cromosoma** (**vector** compuesto por **elementos** a los que llamaremos **genes**). Las **diferentes formas (valores)** que puede tomar un gen se llaman **alelos**.La **posición física** que ocupa cada gen en el cromosoma se llama **lotus**. **Todos los cromosomas** forman la **población** (espacio de soluciones):



**Podemos asumir** que **cada gen indica** si **está presente la feature i con un alelo de 0 o 1**. El **cromosoma** es un **vector de 13 genes**, de los cuales **cada uno indica si contiene o no una de las features** de la matriz de features del DataSet de vinos. La población es el conjunto de cromosomas. Un conjunto de 213 elementos. Dado el enorme volumen de soluciones, los algoritmos genéticos nos dan la posibilidad de una **búsqueda heurística** de la mejor solución.



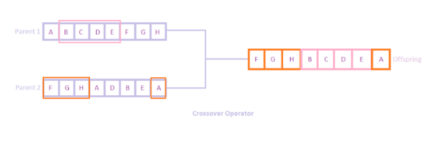
La **población inicial** es un conjunto de individuos de tamaño **size** determinado al azar.

El **Fitness** es una medida de performance del modeloal ajustarlo a los individuos. En modelos de clasificación podría ser accuracy, en modelos de regresión podría ser r2, entre otros. Lo que hace es determinar qué tan apto es un individuo. Nos quedamos con los **n\_best cromosomas**.

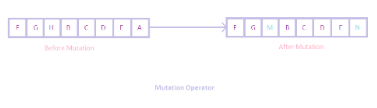
El **Criterio de Terminación** genera una medida del fitness de los cromosomas; un **score**. Es una **medida de error** en caso de que sea menor a un cierto threshold. También puede ser el **número final de iteraciones alcanzada**s.

**Selección:** a los n\_best cromosomas les agregamos n\_rand cromosomas al azar. Algunos se seleccionan para generar descendencia y otros siguen a la etapa de mutación.

**Pairing/Mating:** Selecciona una pareja de cromosomas para aparearlas y generar una nueva descendencia de **n\_children cromosomas**. Se hace mediante **crossover,** combinando genes de ambos individuos:



**Mutation:** Un porcentaje cambia de valor en algunos de sus genes:



**Nueva Generación:** Se evalúan los nuevos cromosomas generados y se seleccionan los n\_best cromosomas. A este proceso se lo llama **elitismo**.

Si bien Python no cuenta con una librería de algoritmos genéticos, es posible usar una implementación simple del algoritmo.

**En Python:**

%run ‘../Notebooks/Genetic.py’

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import StratifiedKFold

est = DecisionTreeClassifier()

cv\_param = StratifiedKFold(n\_splits = 5, shuffle = True)

score = cross\_val\_score(est, X\_sc, y, cv = cv\_param, scoring = ‘balanced\_accuracy’)

print(‘Performance con todas las features: {:.2f}’.format(np.mean(score)))



La implementación tiene los siguientes parámetros:

* **estimator:** Estimador usado para calcular el Fitness.
* **n\_gen:** Número máximo de generaciones (número de iteraciones del algoritmo)
* **size:** Tamaño de la población a considerar.
* **n\_best**: En cada iteración se seleccionan los n\_best cromosomas.
* **n\_rand**:Los n\_rand cromosomas que se seleccionan al azar.
* **n\_children:** Número de nuevos cromosomas que se generan en cada apareamiento.
* **mutation\_rate:** probabilidad de cambiar cada gen espontáneamente.
* **cv**: default 5; argumento para la función cross\_val\_score de sklearn.
* **scoring**: medida de fitness de los cromosomas. “neg\_mean\_squared\_error” para modelos de regresión o “accuracy” para clasificación. Dentro de la clase se multiplica la medida por -1 convirtiéndola en una medida de error.

sel = GeneticSelector(estimator = DecisionTreeClassifier(), n\_gen = 15, size = 200, n\_best = 40, n\_rand = 40, n\_children = 5, mutation\_rate = 0.05, scoring = ‘balanced\_accuracy’)

sel.fit(X\_sc,y)

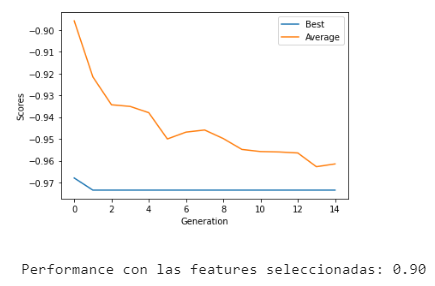


# En el siguiente gráfico Podemos ver cómo fue bajando el error a medida que el algoritmo fue iterando (creando nuevas generaciones). Average es el promedio de los scores de los cromosomas de la generación, y Best es el mejor score de cada generación:

sel.plot\_scores()

score = cross\_val\_score(est, X\_sc[:,sel.support], y, cv = cv\_param, scoring = ‘balanced\_accuracy’)

print(‘Performance con las features seleccionadas: {:.2f}’.format(np.mean(score)))



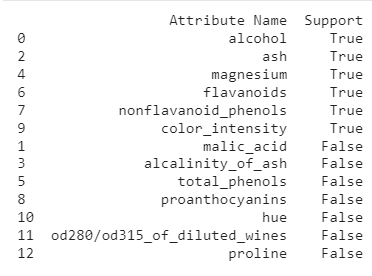
# Vemos finalmente cuáles son las features seleccionadas:

scores = pd.DataFrame()

scores[‘Attribute Name’] = X.columns

scores[‘Support’] = sel.support\_

print(scores.sort\_values(‘Support’, ascending = False))



**Ventajas:**

* **Más rápido y eficiente** que los **métodos tradicionales.**
* **Permite Procesamiento Paralelo.**
* **Genera una lista de “buenas” soluciones.**
* Es **útil** para **grandes espacios de soluciones.**

**Desventajas**:

* **No es apropiado** para **problemas simples**.
* **No garantiza** llegar a la **solución óptima**.
* La evaluación frecuente del fitness lo hace **computacionalmente caro.**

**Conclusiones:**

* Es común encontrar Datasets desbalanceados en la vida real; y es necesario transformarlos para que los algoritmos de Machine Learning funcionen correctamente.
* La **maldición de la dimensionalidad** hace alusión a los problemas que enfrentan los modelos para entrenar cuando tenemos alta dimensionalidad.
* Con **métodos de features selection** podemos reducir los features de los datasets de manera inteligente.
* **Entre los métodos de features selection**, contamos con los **filter methods**, que **rankean las features en función de un valor de importancia**. Muchas veces **necesitan que les definamos un umbral** por debajo del cual las variables dejan de ser importantes.
* Los **wrapper methods** seleccionan **subconjuntos de features** en **función de la performance** que obtienen **al ajustarlos** sobre un modelo.
* Es posible **adaptar los algoritmos genéticos** para **seleccionar las mejores features**.