# **Capítulo 8. Reducción de Dimensiones**

* **Introducción**

Muchos problemas en el mundo de Machine Learning involucran miles o hasta millones de variables por cada set de entrenamiento. Esto, no solo ocasiona que entrenar el modelo sea lento, sino también aumenta la complejidad para encontrar una buena predicción. A este problema se le conoce como la maldición de las dimensiones.

Afortunadamente, existen formas de reducir el número de variables, cambiando de un set de datos imposible de entrenar a uno capaz de ser entrenado.

Como vimos anteriormente, si tienes un modelo clasificador que predice si un paciente pudiera tener diabetes, variables como su signo zodiacal no aportan información relevante para obtener una predicción acertada. Así mismo, dentro de un set de datos, existen variables que se correlacionan directamente. Esto ayuda a disminuir dimensiones porque aquellas variables relacionadas entre sí se pueden mezclar como una sola.

Aunque al disminuir las dimensiones puedes optimizar el entrenamiento de tu modelo, también puedes perder información importante en el proceso. Por lo tanto, se recomienda intentar entrenar los datos completos y ejecutar técnicas de reducción hasta confirmar que es necesario.

Además de ayudar a incrementar la velocidad de entrenamiento, reducir las dimensiones también puede ayudar a la visualización de datos. Cuando se reducen las variables de manera secuencial, puedes ir graficando lo que está sucediendo en cada dimensión y, por lo tanto, tener una mejor percepción de lo que está ocurriendo.

También, puede ayudarte a presentar tus resultados ante otras personas, sobre todo con personas que no están familiarizadas con ML, para tener un acercamiento más amigable del contenido.

En este capítulo hablaremos sobre la maldición de las dimensiones y explicaremos que sucede en espacios de altas dimensiones. También veremos los 2 principales acercamientos a la reducción de dimensiones, proyección y aprendizaje minifold. Por último, aprenderemos a usar 3 de las técnicas de reducción de dimensiones más famosas: PCA, Kernel PCA y LLE.

## **La Maldición de las Dimensiones**

En la vida cotidiana estamos tan acostumbrados a vivir en un mundo tridimensional que nuestra intuición falla al momento de imaginar espacios con dimensiones grandes.Incluso nos cuesta bastante trabajo imaginar un objeto en cuarta dimensión, ni hablar de un elipsoide de 200 dimensiones plasmado en un espacio de 1000 dimensiones.

En la imagen podemos observar que estamos familiarizados con figuras de hasta 3D. Sin embargo, la representación se pone complicada cuando llegamos al objeto en 4D ya que implica movimiento. El comportamiento de los objetos cambia drásticamente en espacios de altas dimensiones.

Chart

Description automatically generated

Por ejemplo, si tomamos un cuadrado como el que teníamos en la imagen de 2D de 1 cm x 1 cm y le marcamos un punto aleatoriamente en su superficie, estadísticamente sólo tendría un 0.4% de probabilidad de estar en un extremo del cuadrado.

Mientras que, si tomamos un hipercubo de 10000 dimensiones y ponemos un punto aleatoriamente, la probabilidad de que caiga en una esquina es de 99.9999%. Esto se debe a que la mayoría de los puntos en un hipercubo están cerca de un borde.

Otra característica cambiante en las altas dimensiones es la distancia. Por ejemplo, si tomamos el mismo cuadrado unitario, podríamos decir que si colocamos 2 puntos de separación podrían tener una distancia promedio de 0.5, mientras que si lo hacemos en un cubo al tener 6 caras la distancia promedio puede aumentar. Digamos que podría ser 0.7.

Si colocamos 2 puntos en un hipercubo de un millón de dimensiones la distancia promedio sería de 408 debido a que en mundos de altas dimensiones hay mucho más espacio disponible. Esto también propicia que en un set de datos con muchas dimensiones, los datos o instancias están muy separadas una de la otra haciendo las predicciones menos confiables. En otras palabras, mientras más dimensiones haya, mayor será el riesgo de un sobreajuste.

En teoría, una posible solución para la maldición de las dimensiones sería aumentar las instancias del set de datos para que haya un balance entre dimensiones y datos. Desafortunadamente en la práctica el número de instancias requeridas para balancear el número de variables crece exponencialmente.

Para darte una idea, con tan solo 100 variables en tu set de datos necesitarías más instancias que los átomos existentes en el universo observable solo para lograr un 0.1 de balance las unas de las otras.

## **Los Acercamientos Más Populares Para La Reducción De Dimensiones**

### **Proyección**

En la mayoría de los casos de la vida real, las instancias de nuestro set de datos no se dispersan uniformemente a través de todas las dimensiones. La mayoría de las variables son casi constantes, mientras que otras están altamente correlacionadas. Como resultado, todas las instancias de entrenamiento caen en un subespacio dimensional más pequeño dentro de un espacio de altas dimensiones.

Chart, scatter chart

Description automatically generated

En la imagen se muestra un set de datos en tres dimensiones representado por círculos. Todos los círculos están dentro de un plano (área gris), el cual puede ser considerado como un plano bidimensional. Si nos quedamos solo con este plano (área gris), obtendríamos un espacio en 2D.

Chart, scatter chart

Description automatically generated

De esta manera estaríamos pasando de un espacio tridimensional a proyectar los puntos en un plano bidimensional. Como tal, ya redujimos la dimensión del problema. Claramente este es un ejemplo sencillo, no siempre será tan fácil hacer las proyecciones, ni será de la misma forma.

Por ejemplo, veamos este set de datos. Las instancias están mayormente distribuidas en el espacio 3D. Cada color representa un tipo de instancias y queremos separarlas por clase. Por lo tanto, necesitamos buscar el acomodo justo, donde al reducir las dimensiones esta división de datos sea lo más clara posible.

Chart, scatter chart

Description automatically generated

Una opción para pasar de 3D a 2D sería proyectar las instancias tal cual en un plano bidimensional. Similar a como si este rollo fuera aplastado por un libro. Haciendo esto, él problema se complicaría más pues se combinan todos los tipos de instancias.

Chart, scatter chart

Description automatically generated

Por lo tanto, en este set de datos no es viable utilizar la reducción por proyección.

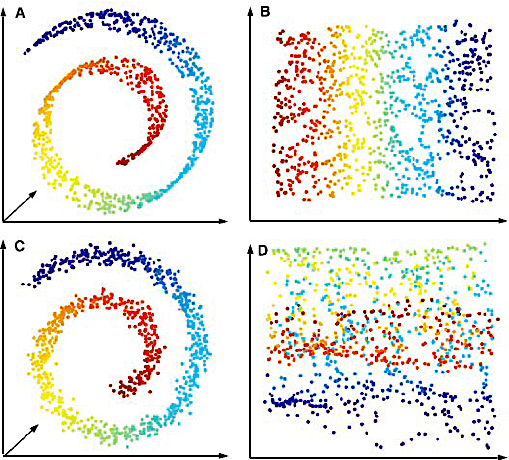
### **Aprendizaje Manifold**

Si retomamos el ejemplo descrito en *reducción por proyección,* llegamos a la conclusión que, al proyectar los datos de manera convencional, el problema se vuelve más complejo porque no se respeta la separación entre los colores. Ante estos sets de datos en forma de rollo se recomienda utilizar algo llamado Manifold Learning o Aprendizaje de Manifold.Básicamenteesto consiste enreducir las dimensiones como si se estuviera desenrollando el set de datos.

Chart, scatter chart

Description automatically generated

Muchos algoritmos de reducción de dimensiones funcionan modelando Manifold donde las instancias de entrenamiento aparentan una dimensión, pero pueden funcionar en otra.



Dentro de la teoría se Manifold, se encuentra una suposición importante en el tema: cualquier tarea será más sencilla de resolver si se expresa en la dimensión baja del Manifold. Sin embargo, este concepto no es siempre correcto.

Por ejemplo, en esta imagen tenemos un set de datos que fue reducido en dimensiones. El clasificador pudo dividir el tipo de instancias con una simple línea a la mitad. En este caso se cumplió la suposición.

Chart, scatter chart

Description automatically generated

Ahora, vamos a ver el siguiente caso. En este ejemplo, el set de datos en tres dimensiones es más fácil de clasificar que el obtenido después de una reducción de dimensiones. Por lo tanto, con este set de datos la suposición no aplica.

Chart, scatter chart

Description automatically generated

En resumen, reducir las dimensiones de tu set de datos antes de entrenar el modelo te va a ayudar a acelerar el proceso de entrenamiento, pero desafortunadamente no siempre te llevará a una solución más simple u óptima. Todo depende del set de datos y no hay una regla generalizada para todos los ejemplos. Es parte del trabajo evaluar qué tipo de modelo es el que te da mejores resultados y poner en balance si vale la pena reducir las dimensiones o no.

## **PCA: Análisis de Componentes Principales**

El Análisis de Componentes Principales o PCA es, por mucho, el método de reducción de dimensiones más popular. Este consiste en identificar el hiperplano que se acerca más a los datos y luego proyectar los datos en él.

### **Conservar la Varianza**

Antes de poder utilizar la proyección para reducir las dimensiones de tus datos, primero necesitas escoger el hiperplano correcto.

Por ejemplo, observemos este ejemplo en donde se intenta reducir dimensiones de 2D a 1D. También se muestran los 3 posibles hiperplanos para hacerlo. Por mucho, la línea continua, es decir, el hiperplano C1 es el que conserva mayor parte de los datos y, por lo tanto, preserva su varianza también.

Chart, line chart

Description automatically generated

Otra forma de justificar esta elección sería elegir el plano que reduzca la distancia media al cuadrado del set de datos original, algo como lo que hemos visto com los errores residuales.

### **Componentes Principales**

El PCA identifica el eje que conserva la mayor varianza posible en comparación al set de datos original. En el caso de la imagen anterior este sería la línea C1, también se prueba con un eje ortogonal a este que sería C2, el cual capta la cantidad de varianza faltante.

En un ejemplo como este que es 2D no hay otra opción, pero si fuera un dataset de mayores dimensiones, el PCA también encontraría un tercer eje, ortogonal a los 2 ejes anteriores, hasta un cuarto o un quinto dependiendo cual sea el total de sus dimensiones.

A cada uno de estos ejes se le conoce como componente principal o PC. En el caso de la imagen anterior el PC1 sería el eje más relevante que corresponde al eje C1, mientras que el PC2 sería el eje C2.

Pero ¿Cómo se pueden encontrar los componentes principales para un set de datos? Afortunadamente hay una técnica de factorización de matrices que se llama descomposición del valor singular o singular value descomposition (SVD).

Esta técnica desintegra la matriz de nuestro set de entrenamiento (X) en una multiplicación de 3 matrices U E VT, donde V contiene vectores unitarios, los cuales a su vez definen los componentes principales que estamos buscando. Básicamente, al descomponer la matriz inicial, nos proporciona los componentes principales de nuestro set de datos.

* + - **PCA con *numpy***

La buena noticia de todo esto es que se puede utilizar *numpy* para realizar esta técnica. Veamos un ejemplo. Empecemos con nuestro set de datos simple para clasificación de candidatos a empleados.

candidates = {'gmat': [780,750,690,710,680,730,690,720,740,690,610,690,710,680,770,610,580,650,540,590,620,600,550,550,570,670,660,580,650,660,640,620,660,660,680,650,670,580,590,690],

'gpa': [4,3.9,3.3,3.7,3.9,3.7,2.3,3.3,3.3,1.7,2.7,3.7,3.7,3.3,3.3,3,2.7,3.7,2.7,2.3,3.3,2,2.3,2.7,3,3.3,3.7,2.3,3.7,3.3,3,2.7,4,3.3,3.3,2.3,2.7,3.3,1.7,3.7],

'work\_experience': [3,4,3,5,4,6,1,4,5,1,3,5,6,4,3,1,4,6,2,3,2,1,4,1,2,6,4,2,6,5,1,2,4,6,5,1,2,1,4,5],

'admitted': [1,1,0,1,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,0,1,0,0,0,0,1,1,0,1,1,0,0,1,1,1,0,0,0,0,1]

}

df = pd.DataFrame(candidates,columns= ['gmat', 'gpa','work\_experience','admitted'])

df.head()

Table

Description automatically generated

Algo muy importante a tomar en cuenta es que PCA asume que el set de datos está centrado a partir del origen de la gráfica por lo que primero tendríamos que centrar los datos restándole la media a los datos originales.

Separamos las variables predictoras de la variable a predecir:

x = df[['gpa','gmat','work\_experience']]

y = df['admitted']

Una vez teniendo nuestros datos centramos x restándole su media:

x\_centrada = x - x.mean(axis=0)

Recordemos que SVD descompone la matriz X en 3 matrices U, E y V por lo que utilizamos la función de *numpy svd.*

U, E, V = np.linalg.svd(x\_centrada)

Y recordemos que la matriz V contiene los vectores con los componentes principales, por lo que para obtener los primeros 2 simplemente transponemos sus primeras 2 columnas.

pc1 = V.T[:, 0]

pc2 = V.T[:,1]

A picture containing text, orange

Description automatically generated

Y listo ya tenemos los 2 componentes principales de nuestros datos.

### **Reducción de Dimensiones**

Una vez que se identifica los componentes principales se puede reducir la dimensión de nuestro set de datos proyectando los datos en el hiperplano conformado por estos componentes principales.

Seleccionar estos hiperplanos nos asegura que nuestra proyección va a conservar la mayor varianza posible.

Para proyectar nuestros datos en el hiperplano y obtener nuestra matriz X de dimensiones reducidas tenemos que *XNUEVA = XW* donde X es nuestra matriz de datos original y W es la matriz que contiene nuestros vectores de componentes principales.

Por lo que si hiciéramos la operación en *Python* tendríamos:

W = V.T[:, :2]

x\_nueva = x\_centrada.dot(W)

x\_nueva.head()

En este caso lo único que hicimos fue tomar los 2 primeros vectores de la matriz V, o los 2 primeros componentes principales y luego los multiplicamos por nuestra X centrada.

Table

Description automatically generated with medium confidence

Como se observa pasamos de tener 3 variables a tener solo 2 variables.

Ahora que sabemos cómo reducir las dimensiones conservando la mayor cantidad de varianza posible, pasemos a nuestro amigo Scikit-Learn.

### **PCA con Scikit-Learn**

En el caso de PCA con Scikit-Learn, este también utiliza la técnica de descomposición SVD para implementarla.

El módulo se llama PCA y lo único que tenemos que hacer es indicarle cuántas dimensiones queremos en nuestros datos con el parámetro *n\_components.*

Otra ventaja de usar PCA con Scikit-Learn es que centra los datos de entrenamiento automáticamente.

Ejemplo:

from sklearn.decomposition import PCA

pca = PCA(n\_components=2)

x\_nueva = pca.fit\_transform(x)

x\_nueva[0:5]

y tan solo entrenando los datos y transformándolo hicimos exactamente lo mismo que habíamos hecho con numpy anteriormente.

Text

Description automatically generated

* + **Distribución de la Varianza**

Otra función bastante útil que nos proporciona Scikit-Learn es la distribución de la varianza conocida como *explained variance ratio*.

Esta distribución nos indica la proporción de varianza que cada componente principal contiene en comparación al set de datos original, para ejecutarlo solo tenemos que utilizar la función *explaines\_variance\_ratio\_,* ejemplo:

pca.explained\_variance\_ratio\_

En este caso tenemos que el PC1 mantuvo casi el 100% de la varianza de los datos mientras que el PC2 no mantuvo prácticamente nada de la varianza, por lo que asumimos que el PC2 no contiene demasiada información.

### **Elegir el Número Correcto de Dimensiones**

En vez de elegir arbitrariamente el número de dimensiones al cual queremos llegar, es más fácil identificar el número de dimensiones que suman una proporción suficientemente de varianza en comparación a nuestro set de datos original.

Esto suponiendo que quieres reducir dimensiones para aumentar la efectividad de tu modelo.

Si lo que se desea es reducir dimensiones para visualizar datos entonces tendrías que reducirlo a 2 o 3 dimensiones.

La forma más fácil y útil para indicarle a tu modelo PCA la varianza que quieres conservar al reducir dimensiones es con el parámetro *n\_components*, pero en vez de indicarle el número de dimensiones que deseas, el algoritmo reconoce los números entre 0 y 1 como porcentaje de varianza deseada.

Para ejemplificar esto, vamos a crear un modelo donde deseamos al menos el 90% de la varianza conservada.

pca = PCA(n\_components=0.90)

x\_nueva = pca.fit\_transform(x)

x\_nueva[0:5]

Text

Description automatically generated

En este caso parece que la dimensión óptima para nuestro modelo es de 1 sola dimensión, lo cual tiene sentido ya que antes vimos que el PC1 mantiene casi el 100% de la varianza de los datos originales. Entonces podemos pasar de tener 3 variables a solo utilizar 1.

### **PCA Para Compresión**

Después de reducir las dimensiones, el set de datos toma mucho menos espacio. Como vimos antes pasamos de tener 3 variables a tener solo 1, pero habrá casos donde tendremos miles de variables y pasaremos a tener cientos.

Así cómo es posible hacer la compresión de los datos con Scikit-Learn-Learn, también es posible descomprimirlos con esta misma función. Esto se hace aplicando la inversa de la proyección del PCA. Desafortunadamente al hacer este proceso, habrá pérdida de información. Por lo tanto, el set de datos que arroja la computadora al final del proceso no será igual al original pero sí uno muy parecido.

Para descomprimir los datos utilizaremos la función *inverse\_transform*:

pca = PCA(n\_components=1)

x\_nueva = pca.fit\_transform(x)

x\_recuperada = pca.inverse\_transform(x\_nueva)

x\_recuperada[0:5]

Text

Description automatically generated

De esa manera regresamos a las 3 variable originales y de hecho las cantidades son bastante parecidas a los datos originales.

Table

Description automatically generated

### **PCA Aleatorio**

PCA tiene otro parámetro que se llama *svd\_solver*, si este parámetro se cambia a *randomized*,, Scikit-Learn usa un algoritmo estocástico , el cual encuentra rápidamente vectores que se aproximan a los componentes principales. Este método es útil cuando se requiere reducir la memoria computacional ya sea porque se está trabajando con un set de datos grande o porque simplemente se quiere gastar menos recursos computacionales.

Ejemplo:

pca = PCA(n\_components=1, svd\_solver="randomized")

x\_nueva = pca.fit\_transform(x)

x\_nueva[0:5]

Text

Description automatically generated

De igual manera, Scikit-Learn cambia a este método si detecta que se tienen más de 500 variables o se requiere menos del 80% de la proporción de la varianza. Si es al contrario el método tradicional normalmente resulta en mejores resultados.

### **PCA Incremental**

Dentro de los métodos de PCA que hemos revisado hasta el momento, todos requieren que el set de datos esté en la memoria para poder implementarse.

Ante esto, se desarrollaron algoritmos como el **PCA incrementales** que permiten dividir el set de datos en pequeños sub-sets y así ir alimentando el algoritmo paulatinamente. Esto puede ser bastante útil si se tienen sets de datos muy grandes o si se quiere optar por aprendizaje online.

Para utilizar este tipo de algoritmo tenemos el módulo de Scikit-Learn-Learn *IncrementalPCA*

La diferencia con este módulo es que se debe de aplicar la función *partial\_fit* para cada uno de los sub-sets que utilicemos.

Ejemplo con 3 sub-sets:

from sklearn.decomposition import IncrementalPCA

subsets = 3

ipca = IncrementalPCA(n\_components=1)

for subset in np.array\_split(x, subsets):

ipca.partial\_fit(subset)

x\_nueva = ipca.transform(x)

x\_nueva[0:5]

Como vemos lo único que hicimos fue seleccionar el número de sub-sets que deseamos, dividimos *x* en ese número con la función de *numpy Split*, y entrenamos el modelo con cada uno de los sub-set generados.

## **Kernel PCA**

El truco de Kernel para PCA funciona igual que el truco de Kernel para SVM (capítulo 5). Esto le permite ejecutar reducciones no lineales con alta complejidad, justo como permitía utilizar SVM en modelos complejos. Vamos a ver un ejemplo de cómo utilizar el truco de Kernel en este caso:

from sklearn.decomposition import KernelPCA

pca\_rbf = KernelPCA(n\_components=2, Kernel="rbf")

Como vemos al igual que en SVM solo hace falta especificar con que Kernel queremos modelar nuestro algoritmo. En la siguiente imagen se ejemplifica las diferencias entre los diferentes tipos de Kernel: Kernel lineal o un PCA convencional, Kernel RBF y otro Kernel muy popular para PCA llamado sigmoide.

Chart, scatter chart

Description automatically generated

* + **Seleccionar Un Kernel Y Afinar Sus Hiperparámetros**

Este tipo de algoritmo es no supervisado, lo que significa que no hay una medida de desempeño predeterminada. Dicho esto, podemos asumir que la reducción de dimensiones es una preparación para después ejecutar el aprendizaje supervisado, ya sea clasificación o regresión.

Para afinar tu modelo podemos usar el método grid search. Como lo vimos en el capítulo 2, esta herramienta consiste en probar varios valores para los hiperparámetros en un pipeline y después nos indica cual fue el hiperparámetros con mejor desempeño.

Para visualizar esto, vamos a ver un ejemplo. Por medio de un grid search vamos a probar 10 diferentes gammas que oscilan desde los 0.01 a 0.05 y los 2 Kernel antes mencionados RBF y sigmoide. Nuestro modelo de clasificación será la regresión logística.

from sklearn.model\_selection import GridSearchCV

from sklearn.linear\_model import LogisticRegression

from sklearn.pipeline import Pipeline

clf = Pipeline([

("pca", KernelPCA(n\_components=2)),

("reg", LogisticRegression())

])

Una vez que realizamos el pipeline con la reducción PCA y establecimos el modelo clasificador. vamos a configurar los parámetros del grid search, para los valores numéricos creamos un vector con la función de *numpy linspace.*

grid = [{

"pca\_\_gamma": np.linspace(0.1,0.5,10),

"pca\_\_Kernel": ["rbf","sigmoid"]

Ejecutamos el grid search:

search = GridSearchCV(clf, grid)

search.fit(x,y)

y, por último, ejecutamos la función *best\_params* para saber cuáles son los mejores hiperparámetros:

print(search.best\_params\_)



Como vemos la mejor combinación de hiperparámetros para este modelo sería una gamma de 0.14 y un Kernel de tipo “*RBF*”.

Otra forma de medir el desempeño del Kernel PCA es comprimir los datos, luego descomprimirlos y comprar los datos descomprimidos con los datos originales. En el caso del Kernel PCA es importante colocar el parámetro *fit\_inverse\_transform* como “*True*”, ya que de otra manera no podremos utilizar esta función.

Vamos a poner un ejemplo:

rbf = KernelPCA(n\_components=1, Kernel="rbf", gamma=0.02, fit\_inverse\_transform=True)

x\_comprimido = rbf.fit\_transform(x)

x\_descomprimido = rbf.inverse\_transform(x\_comprimido)

Una vez tenemos todo preparado solo utilizamos la función *mean\_squared\_error* para comparar los datos descomprimidos con los datos originales:

from sklearn.metrics import mean\_squared\_error

mean\_squared\_error(x, x\_descomprimido)



De esta manera ya tenemos el error calculado y conocemos otra manera de medir el desempeño para PCA.

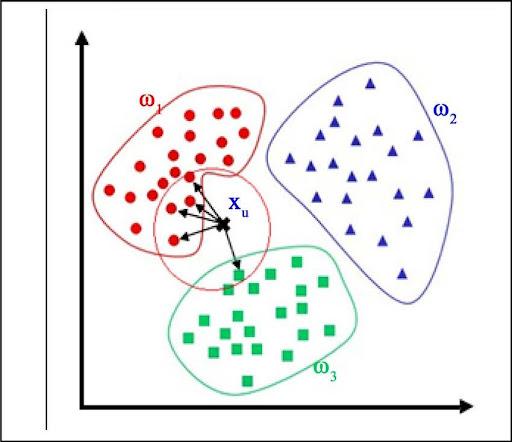
## **LLE- Incrustación Lineal Local**

LLE es otro algoritmo poderoso para reducir dimensiones no lineales. Es una técnica de tipo aprendizaje manifold que no se basa en proyecciones como otros algoritmos que hemos visto.

LLE funciona, primero que nada, midiendo como cada instancia de entrenamiento se relaciona linealmente con sus instancias vecinas.Después de hacer esto LLE busca una representación lineal de pocas dimensiones del set de datos donde estas relaciones entre instancias cercanas o vecinas están bien preservadas.Este método funciona bien para desenrollar minifolds, especialmente si no tienen mucho ruido en los datos.

Para utilizar el LLE con Scikit-Learn necesitamos el módulo *LocallyLinearEmbedding,* una vez importando el módulo, tenemos que seleccionar el número de dimensiones, igualmente con *n\_components* y otro parámetro bastante importante es el número de vecinos*, n\_neighbors*. Este parámetro nos sirve para indicar con cuántos vecinos queremos que se relacione cada instancia.

Lo que hace el algoritmo es un sistema de optimización donde intenta reducir las dimensiones, pero manteniendo las características de cada vecino en cada instancia.



Por ejemplo, la instancia x tiene de vecinos tanto instancias de clase verde como de clase roja, por lo que al reducir dimensiones se intentan preservar esas características.

Esto el algoritmo lo logra creando una matriz de pesos donde al multiplicar por la matriz de datos original, este tiene menos dimensiones y cada instancia mantiene la relación con las instancias de su alrededor.

Vamos a hacer una prueba con 5 vecinos.

from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n\_components=2, n\_neighbors=5)

x\_nueva = lle.fit\_transform(x)

x\_nueva[0:5]

Text

Description automatically generated

De esta manera tenemos nuestras dimensiones reducidas con LLE.

Si se tuviera un set de datos enrollado, utilizando el método LLE quedaría la dimensión como se muestra a continuación:

Chart, scatter chart

Description automatically generated Chart, scatter chart

Description automatically generated

En general podemos decir que se hizo un buen trabajo desenrollando los datos, aunque del lado derecho quedaron bastante apretados.

Un problema con LLE es que en realidad tiene bastante complejidad computacional, por lo que no es muy recomendable usarlo si se tiene un set de datos muy grande.

## **Otras Técnicas Para Reducir Las Dimensiones**

Hay muchas otras técnicas para reducir dimensiones, aunque son menos utilizadas que PCA, la mayoría de ellas también están disponibles en Scikit-Learn.

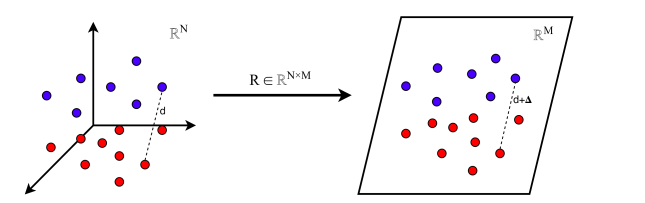
* + - Proyecciones Aleatorias:

Como lo dice su nombre, esta técnica proyecta un espacio de bajas dimensiones usando una proyección lineal aleatoria.

El truco para que funcione esta técnica depende del número de instancias en los datos y el objetivo de dimensión, para más información puedes checar el documento de Scikit-Learn sobre random projection.

* + - * Material

[https://Scikit-Learn-learn.org/stable/modules/random\_projection.html](https://scikit-learn-learn.org/stable/modules/random_projection.html)



* + - **Escalado multidimensional:**

Este método reduce las dimensiones, intentando mantener la distancia entre cada una de las instancias.

Chart

Description automatically generated

* + - **IsoMapa:**

Este método también funciona principalmente con los vecinos de cada instancia, mientras intenta mantener la distancia con cada uno de ellos.

Chart, scatter chart

Description automatically generated

* + - **Incrustación de vecinos estocástica con distribución t. t-SNE**

Este tipo de técnica se usa principalmente para visualización ya que junta instancias con características similares y separa instancias con características diferentes, por lo que es ideal para graficar modelos de clasificación.

Chart, scatter chart

Description automatically generated

* + - **Análisis discriminante lineal LDA.**

Este es un algoritmo de clasificación, aunque durante el entrenamiento aprende sobre los ejes más discriminantes entre clase y clase. Una vez hecho esto define un hiperplano para dividir las clases en los datos.

El gran beneficio sobre este método es que intentara mantener el límite entre clases lo más lejos posible, como los SVM con margen grande. Si ya se tiene en mente la reducción de dimensiones desde el inicio se recomienda utilizarlo antes que algún modelo de solo clasificación.