# **Capítulo 4. Modelos De Entrenamiento**

## **Introducción**

Hasta el momento hemos utilizado modelos de entrenamiento para ML, pero no hemos profundizado en la mayoría de estos.

Si analizamos los ejercicios de capítulos anteriores, te sorprenderás de cuanto se puede hacer sin saber de verdad en que consiste la funcionalidad de los modelos.

Optimizamos una regresión lineal, creamos un clasificador de medicamentos, y todo esto sin siquiera saber de qué se trata cada modelo.

Aun así, tener un buen entendimiento de cómo funcionan las cosas te va a ayudar a identificar cada vez más rápido cual es el modelo que necesitas.

Entender que hay por debajo de los algoritmos, te ayudara a entender que hiperparámetros usar, que tipo de análisis de error utilizar, encontrar bugs y arreglarlos, saber qué tipo de procesamiento de datos te conviene utilizar.

Aunque en este capítulo vamos a profundizar en el entendimiento, la construcción y el entrenamiento de redes neuronales. También vamos a profundizar en el modelo de regresión lineal, uno de los modelos más simples que hay.

Veremos que hay 2 maneras distintas de entrenarlo:

* Una usando la forma cerrada que consiste en encontrarlos parámetros que mejor se ajustan al modelo y que minimizan la función de costo.
* Usar una optimización iterativa que se llama gradiente descendente, y que poco a poco va ajustando los parámetros para minimizar la función de costo y que eventualmente va a converger en los mismos parámetros que el primer método.
* Para utilizar el método del gradiente existen diferentes técnicas como batch, mini-batch y estocástico.
* También se explicará en qué circunstancias conviene utilizar cada uno.

Después de eso vamos a profundizar en el tema de la regresión polinomial, un modelo más complejo que se puede ajustar a data sets no lineales.

Veremos que como este modelo tiene parámetros que pertenecen también a la regresión lineal, puede que tienda a cometer overfitting, así que profundizaremos en métodos para saber si es que el modelo está cometiendo overfitting o no.

Usando curvas de aprendizaje, y luego aprendiendo algunas medidas de regularización las cuales ayudan a reducir el riesgo de overfitting.

Por último, profundizaremos en 2 modelos que normalmente se utilizan para clasificación, regresión logística y regresión softmax.

## **Regresión Lineal**

En el capítulo 1 vimos una simple demostración de un modelo de regresión del índice de pobreza:



Donde PR es el índice de pobreza, IPC es el ingreso per cápita, m y b son los parámetros que queremos encontrar para ajustar nuestra regresión.

En términos generales, un modelo lineal genera predicciones al hacer una suma ponderada de sus variables, más una constante llamada bias.

A picture containing diagram

Description automatically generated

En la ecuación:

* y es la variable para predecir, por ejemplo, el índice de pobreza.
* n es el número de variables predictoras, por ejemplo, el ingreso per cápita en el modelo pasado.
* Las x representan el valor de las variables predictoras.
* Las b son los parámetros del modelo, los cuales son los que intentamos ajustar para reducir el error o la distancia entre los valores y b0 es el parámetro conocido como bias.

Si usamos nuestros conocimientos de algebra lineal, podríamos reescribir esta ecuación de una manera vectorizada lo que la haría mucho más concisa.

Recordemos que un vector puede contener múltiples valores dentro, por lo que podemos hacer una multiplicación de vectores.

Text

Description automatically generated

En la ecuación:

* Donde b es el vector de parámetros del modelo, contiene el bias b0 y los parámetros de peso de los predictores b1 a bn.
* Donde x es el vector con los valores de nuestras variables predictoras y donde se le asigna un multiplicador al bias x0 que siempre será igual a 1.
* En este caos b\*x es el producto punto de los vectores b y x, lo que sería igual a b0x0+b1x1+b2x2…. Un ejemplo de cómo se vería una multiplicación de producto punto es este: Diagram

  Description automatically generateddonde los valores se multiplican por su mismo índice, podríamos decir que este producto puto también es igual a bT\*x, donde transponemos el vector b.
* Por último, hb es la función de hipótesis donde utilizamos los posibles valores diferentes para los parámetros b.

Ahora que conocemos en que consiste un modelo lineal, te preguntaras ¿Cómo lo entrenamos?

Bueno, recuerdas que entrenar un modelo significa afinar los parámetros para que el modelo se ajuste lo mejor posible al set de entrenamiento.

Para hacer esto primero tenemos que medir que tan bien o tan mal se ajusta el modelo al set de entrenamiento, antes vimos que una de las medidas más populares para modelos de regresión es el RMSE.

Por lo que en este caso para entrenar el modelo tenemos que encontrar un conjunto de parámetros b, el cual minimice el RMSE al máximo.

En la práctica es más fácil minimizar el MSE que es el error medio cuadrado ya que el RMSE es la raíz cuadrada del MSE, por lo que, si minimizas el MSE, lo único que tienes que hacer es sacarle la raíz cuadrada para saber cuál es el RMSE.

La ecuación del MSE para una regresión lineal se puede definir de la siguiente manera



Si dividimos la ecuación en partes:

* La m es el número de muestras o eventos que tenemos en nuestro set de datos.
* Entonces de 1 a el número de muestras vamos a calcular el error cuadrado, que en este caso es, nuestro vector de parámetros multiplicado por nuestro vector de predictores menos el valor de la variable a predecir, todo esto al cuadrado.
* Al final dividimos esta sumatoria entre el número de eventos.
* Nuestro error depende de los parámetros que utilicemos por lo que el MSE depende del vector de parámetros b.

### **La Ecuación Normal**

Para encontrar el valor de b que minimiza la función de costo, en este caso MSE existe una solución de tipo closed-form o forma cerrada. En otras palabras, una ecuación matemática que regresa un resultado directamente.

Esta se conoce como la ecuación normal:



En esta ecuación:

* b es nuestro vector de parámetros, como podemos observar el objetivo de la ecuación es encontrar el mejor conjunto posible de estos.
* X es nuestro vector de predictores, como ya lo habíamos mencionado antes, nunca hay que olvidar agregar un valor x0 que sea igual a 1 y sirva de multiplicador para el bias.
* Y es el vector que contiene los valores que queremos predecir, en el ejemplo pasado era el índice de pobreza.

Vamos a crear nuestro propio set de datos en Python y ver cómo funciona la ecuación.

Para esto utilizaremos la función de numpy random, la cual genera valores aleatorios, y con esta vamos a generar un set de datos con tendencia lineal.

import numpy as np

import matplotlib.pyplot as plt

x = np.random.rand(100,1)

Empezamos con el vector x, el cual consiste en 100 valores aleatorios que van del 0 al 1, ya que la función rand recibe el primer parámetro que es el número de valores y en el segundo parámetro el valor máximo.

Text

Description automatically generated

Para que los datos tengan una tendencia lineal, nuestro vector y, va a tener la forma de la ecuación y = mx + b, solo que en este caso le añadiremos un poco de ruido, sumándole un valor aleatorio de igual manera de 0 a 1.

y = 3 + 3 \* x + np.random.rand(100,1)

En este caso esto corresponde a y = 3+3x + (valor aleatorio de 0 a 1), ya que si no agregamos estos valores la ecuación seria completamente lineal y no emularíamos a un ejemplo de verdad donde habrá algo de ruido.

Text

Description automatically generated

Ahora lo graficamos:

plt.scatter(x,y)

plt.show()

Chart, scatter chart

Description automatically generated

Como podemos observar tenemos un set de datos que sigue una tendencia lineal.

Ahora vamos a ejecutar nuestra ecuación normal, para esto necesitamos aplicar ciertos términos de algebra lineal como la inversa, transpuesta y producto punto antes mencionado.

Si no se recuerda cómo funcionan estos conceptos en este [video](https://www.youtube.com/watch?v=3BpGef99HEs), se explica muy claro y fácil en tan solo 10 minutos. Una vez habiendo repasado esto estarás feliz de saber que no tenemos que hacer todos esos cálculos a mano.

Numpy tiene un módulo de algebra lineal con el que se pueden hacer todas estas operaciones sin mayor complejidad.

En este caso para la ecuación normal necesitamos la inversa que sería la función **inv()**, la transpuesta que sería el método T y el producto punto que sería la función **dot()**.

Antes de hacer esto hay que recordar que hay que crear un x0, que tiene que ser un 1.

En este caso vamos a crear un vector que contenga todos los valores de x y cada valor de x este acompañado de un 1.

Para esto utilizaremos la función de numpy c\_ la cual sirve para crear sablistas, en este caso vamos a crear una lista que este llena de las sablistas con los valores de x y un 1.

Para crear los 1 usaremos la función de numpy ones, la cual sirve para crear vectores o matrices de 1.

x\_b = np.c\_[np.ones((100,1)),x]

x\_b

A picture containing table

Description automatically generated

Ahora si procedemos a ejecutar la ecuación normal.

param = np.linalg.inv(x\_b.T.dot(x\_b)).dot(x\_b.T).dot(y)

param



ahí tenemos nuestro resultado, recordemos que nuestra ecuación original era y = 3 + 3x + ruido de los numero aleatorios.

Es por este ruido que con esos parámetros la ecuación quedaría de esta manera y = 3.52 + 2.95x, ambos números están cerca del 3 por lo que no es un mal acercamiento.

Ahora con estos nuevos parámetros podremos realizar algunas predicciones.

Como podemos ver en nuestra gráfica, tenemos una x que va de 0 a 1, por lo que vamos a crear un vector que va de 0 a 1 con la función de numpy array.

x\_nuevo = np.array([[0],[1]])

x\_nuevo



Después vamos a agregarle x0 que es igual a 1 como en nuestro x anterior.

x\_nuevo\_b = np.c\_[np.ones((2,1)),x\_nuevo]



Por último, vamos a hacer las predicciones multiplicando este nuevo vector por nuestro vector de parámetros que hicimos anteriormente.

y\_pred = x\_nuevo\_b.dot(param)

y\_pred

Text

Description automatically generated

Esta es la predicción que nos dan nuestros parámetros para x cuando es 0 y x cuando es 1.

Vamos a graficar nuestras predicciones con una línea roja, frente a nuestros datos en puntos azules, para ver que tanto se ajustan a la realidad.

plt.plot(x\_nuevo,y\_pred,"r-",label="Predicción")

plt.scatter(x,y,label="Datos")

plt.legend()

plt.show()

Chart, scatter chart

Description automatically generated

Como podemos observar la línea de predicción parece que se ajusta bastante bien a la tendencia de los datos.

También podemos realizar regresiones lineales con scikit-learn, para esto solo necesitamos importar su módulo de **LinealRegression**.

from sklearn.linear\_model import LinearRegression

Para utilizar el módulo solo tenemos que crear el objeto y entrenar los datos con la función fit.

reg\_lin = LinearRegression()

reg\_lin.fit(x,y)

en este momento ya tenemos nuestra regresión lineal hecha, si queremos ver cuáles son los parámetros que nos asigna scikit, solo tenemos que usar los métodos **intercept\_** y **coef\_.**

reg\_lin.intercept\_,reg\_lin.coef\_



Ahí tenemos los resultados y cómo podemos ver fue bastante sencillo obtenerlos con el módulo de scikit.

Hay una cuestión cuando usas la librería de scikit, y esta es que se basa en una función de scipy que se llama lstsq mejor conocida como least squares o mínimos cuadrados.

La función de scipy hace esto b = x+y donde x+ es la pseudoinversa de x.

La pseudoinversa se utiliza usando la técnica para factorización de matrices SVD o **singular value descomposition** en español descomposición del valor único. Utilizar la pseudo inversa es más eficiente que la ecuación normal computacionalmente hablando, ya que se puede dar el caso en que las matrices no sean invertibles, mientras que la pseudoinversa siempre se puede definir.

### **Complejidad Computacional**

Como vimos anteriormente la ecuación normal necesita calcular la inversa de xTx. esto puede sugerir una gran complejidad en la computadora ya que mientras más variables se le agregue al problema, la memoria y capacidad computacional necesaria para efectuar la ecuación crece exponencialmente.

Cuando se quiere usar un modelo de regresión lineal usando la ecuación normal o alguna otra, lo usual es que los cálculos sean bastante rápidos, aunque la complejidad computacional es lineal en cuanto al número de variables a las que se quiere predecir. En otras palabras, si se hacen predicciones si se agrega el doble de variables, la computadora va a tardar el doble de tiempo en ejecutarlas.

Para evitar tener que saturar tanto la carga computacional en caso de que se tenga un gran número de variables, vamos a ver otro método para hacer regresiones lineales.

## **Descenso de Gradiente**

### **Suma de Residuos Cuadrados**

Antes de explicar en qué consiste y cómo funciona el descenso de gradiente, tenemos que entender lo que es la suma de residuos cuadrados.

Como vimos en la regresión lineal, al hacer una predicción sobre una serie de datos, esta intenta adaptarse lo mayor posible, aunque lo más común es que quede cierta diferencia entre la predicción y los datos.

A estas diferencias se les llaman residuales.

Chart, line chart

Description automatically generated

Como se puede ver en la imagen tenemos una regresión donde la línea azul es la función predictora, los puntos azules son las predicciones, los puntos morados son los datos reales, y las líneas rojas es la diferencia entre las predicciones de nuestro modelo y los datos reales (residuales).

Vamos a calcular el valor de nuestros residuales, para hacerlo solo necesitamos restar el valor de nuestros datos (puntos morados) del valor de nuestra predicción (puntos azules).

Text

Description automatically generated

Una vez tenemos nuestros 5 residuales calculados, lo único que resta hacer es calcular la suma de residuales cuadrados, que como su nombre los dice consiste en elevar todos los residuales al cuadrado y sumarlos.



Listo ya tenemos nuestra suma de residuales cuadrados que es 2.4.

Una vez sabiendo eso, una técnica utilizada para hacer la regresión es cambiar el valor donde intercepta la línea, para ver si se ajusta mejor a los datos.

Por ejemplo, en esta regresión tenemos una función de y = 0.7x + 0.7, en las funciones lineales y = mx + b, el parámetro b es el que delimita donde intercepta la línea, en la gráfica.

Chart, line chart

Description automatically generated

Como podemos observar con y = 0.7x + 0.7, la línea hace la intercepción en el eje vertical en el valor 0.7.

Pero qué pasa si cambiamos la intercepción a 1 con y = 0.7x + 1.

Chart, line chart

Description automatically generated

Como vemos la intercepción de la línea subió hasta el valor 1 por lo que ahora tenemos predicciones diferentes.

Vamos a calcular el nuevo valor de los residuales.

Text

Description automatically generated

Ya teniendo el valor de los residuales calculamos el SRC de nuevo:



Y vemos que en este caso con y = 0.7x + 1, tenemos un valor de residuales cuadrados más alto, por lo que hay un mayor error en esta función lo que significa que y = 0.7x + 0.7 se ajusta mejor a los datos.

Si probamos con muchos valores diferentes en el valor de intercepción de la función y graficamos la suma de sus residuales cuadrados vamos a tener algo que se ve como esto:

Chart, scatter chart

Description automatically generated

Así se van viendo en la gráfica de todos los errores con los diferentes valores en el parámetro de intercepción de la ecuación.

Como podemos ver efectivamente 0.7, tiene el valor más bajo de error, cualquier otro valor en la intercepción sea más alto o bajo va a empezar a incrementar la suma de residuales cuadrados.

Es obvio que estar calculando el error para cada posibilidad de intercepción en la gráfica es algo bastante tardado y nada optimo, y es ahí donde entra el descenso de gradiente, el cual vuelve este procedimiento mucho más eficaz y rápido.

### **Descenso de Gradiente Batch**

Una vez que sabemos cómo funciona la suma de residuales al cuadrado podemos pasar a ver cómo funciona el descenso de gradiente.

Empecemos desde el punto de partida de la ecuación de SRC.



Chart, scatter chart

Description automatically generated

Como vemos la instancia A, tiene un valor de 2, mientras que la predicción tiene un valor de 1.4.

Sabemos que nuestra función de predicción es y = 0.7x + 0.7, por lo que el modelo en esa instancia lo calculo como 1.4 = 0.7(1) + 0.7.

Esto significa que si queremos encontrar la suma residual cuadrada para cada valor posible de b en y = mx + b, podemos crear una función de este modelo en específico.



Donde x es nuestra variable predictora y b es nuestra variable para ajustar la intersección de la recta.

Entonces:



Si graficamos esa ecuación:

Chart, line chart

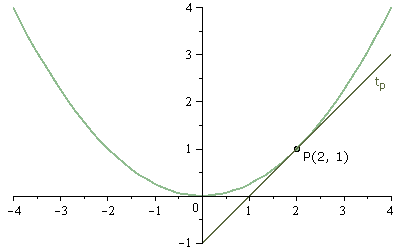
Description automatically generated

De esta manera ya sabemos cuál va a ser la suma de residuales cuadrados para todos los valores posibles del parámetro b, sin tener la necesidad de estar calculando uno por uno.

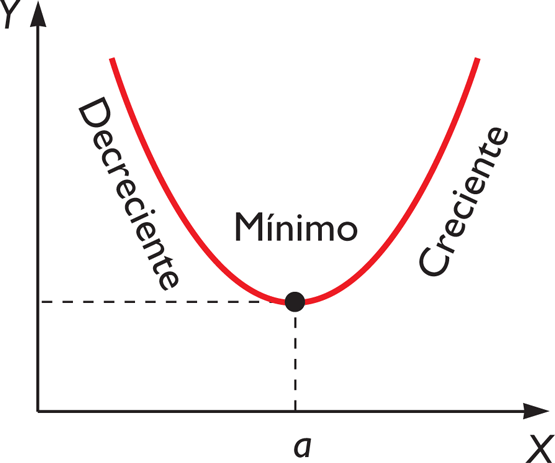
Una vez teniendo esta función es donde el tema se pone un poco más matemático ya que introduciremos el concepto de la derivada.

Recordemos que la derivada de una función nos ayuda a conocer la sensibilidad de cambio en cierto punto de la función.

En el caso de nuestra función de la suma de residuales cuadrados, nos proporciona la recta tangente o la pendiente en este caso para un punto en la función.



Esto es increíblemente eficaz para problemas de minimización, ya que al conocer la pendiente en cada punto podemos saber si está creciendo (positiva) o disminuyendo (negativa), y cuando lleguemos a un punto donde la pendiente será 0, significando que hemos encontrado un valor mínimo.



Entonces básicamente, el descenso de gradiente utiliza la derivada parcial de nuestra función de la suma de residuales cuadrados para ir calculando la pendiente de esta curva, hasta que se convierta en 0.



La verdad que calcular una derivada parcial no es un procedimiento muy complejo, si requieres repasar como se hace, en este [video](https://www.youtube.com/watch?v=tb00qQBYm48) se explica bastante bien.

Para calcular la derivada de nuestra ecuación:



Hay que utilizar la regla de la cadena en cada una de las operaciones



Después de aquí es donde entra el concepto de learning rate o ritmo de aprendizaje el cual se explica a fondo más adelante.

Basta con decir que el learning rate nos sirve para decir que tan rápido queremos avanzar en nuestras pruebas de encontrar la pendiente optima calculando la derivada.

Supongamos que nuestro learning rate es de 0.1.

Y nuestra primera derivada parcial con un numero aleatorio de 0.3 de intercepto b nos da un resultado de una pendiente de -2.

Entonces tenemos que multiplicar el valor de la pendiente por nuestro learning rate.

Paso = -2 \* 0.1 = -0.2

Este resultado es el tamaño que daremos para nuestro siguiente paso ya que este se lo restamos a la pendiente que utilizamos.

b = 0.3 – (-0.2) = 0.5

En este caso ahora tenemos que calcular la derivada de nuestra función cuando el intercepto b es 0.5.

Supongamos que nos da un resultado de -1.8.

Entonces el paso = -1.8 \* 0.1 = -0.18

Por lo que b = 0.5 – (-0.18) = 0.68

Recordemos que nosotros de antemano ya sabemos que el mínimo está en 0.7, aunque muchas veces no se sabrá dónde está el mínimo, es por eso que tenemos que hacer uso de las derivadas parciales y el descenso de gradiente.

Cuando el gradiente se acerca al punto mínimo los pasos se vuelven mucho más pequeños.

Por ejemplo, sabemos que la pendiente en 0.68 esta casi en el mínimo por lo que será muy pequeña, digamos que es 0.05.

Entonces el paso será de = 0.05 \* 0.1 = 0.005.

Y la nueva pendiente 0.68 – (-0.005) = 0.685

Hay que tomar en cuenta que en estos modelos normalmente se utilizan un mínimo de 1000 pasos, por lo que va a llegar un momento en el que la pendiente llegue a 0.

Por lo que el paso = 0 \* 0.1 = 0

Y el valor de intercepto b = 0.7 \* 0 = 0.7

Esto es lo que nos asegura que una vez que llegamos al valor mínimo de la función de suma de residuales cuadrados, este se va a quedar ahí.

Diagram

Description automatically generated

Un parámetro muy importante para el gradiente descendente es el tamaño de sus pasos, en otras palabras, que tan rápido queremos que baje.

Esto es importante porque si vamos muy lento, puede que nos tome tanto tiempo que nunca lleguemos al valor mínimo.

Si vamos demasiado rápido puede que nos pasemos el valor mínimo sin darnos cuenta.

A este hiperparámetros se le conoce como learning rate o ritmo de aprendizaje.

Chart

Description automatically generated

Como podemos ver en la imagen, con un ritmo de aprendizaje lento, el sistema ya hizo bastantes iteraciones y no ha llegado al punto mínimo de la curva.

Con un ritmo de aprendizaje optimo, el sistema no hace tantas iteraciones y llega al fondo de la curva.

En el caso del ritmo de aprendizaje rápido, este tenía tanta velocidad que se saltó el fondo de la curva, terminando incluso en un peor lugar que donde comenzó.

Hay que tomar en cuenta que no todas las funciones de costo se ven como parábolas donde se llega al fondo solo avanzando hacia abajo.

En algunas ocasiones puede haber más curvas, hoyos y terrenos más difíciles donde se puede complicar converger al mínimo.

Cuando se tienen varias curvas el gradiente puede llegar al llamado mínimo local, más puede no encontrar el mínimo global.

Diagram

Description automatically generated

Por ejemplo, en la imagen anterior si el gradiente empieza por la derecha, puede que encuentre el mínimo local, aunque tal vez no encuentre el mínimo global que es más optimo.

Afortunadamente la función de costo MSE es un tipo de función convexa, lo que significa que cuando seleccionas dos puntos en la curva de la función y trazas una línea en estos dos puntos, la línea nunca va a cruzar la curva.

Esto significa que no hay mínimos locales, solo un mínimo global.

El MSE también es una función continua cuya pendiente nunca cambia abruptamente. Estos dos factores garantizan que el gradiente descendente siempre se puede acercar al mínimo global. Asumiendo que se espera el tiempo necesario y el ritmo de aprendizaje no es muy alto.

De hecho, la función de costo tiene la forma de un tazón o una parábola en 3D, aunque puede ser un tazón muy alargado si nuestras variables tienen escalas muy diferentes.

Diagram

Description automatically generated

Como podemos ver en la imagen a, en los ejes x y y las variables tienen escalas diferentes, por lo que el gradiente tiene dificultad para llegar al mínimo.

Mientras que en la imagen b las variables tienen la misma escala, por lo que el gradiente va directamente al mínimo.

Claramente podemos asumir que cuando trabajemos con gradiente descendiente lo más recomendable es escalar todas las variables.

Este diagrama también demuestra que entrenar un modelo significa buscar una combinación de parámetros que minimiza la función de costo.

Es una búsqueda en el espacio de parámetros del modelo, entre más parámetros tenga el modelo mayor, el mayor número de dimensiones este espacio tiene.

Por lo que obviamente entre más dimensiones hay, más complejo es encontrar el mínimo. No es lo mismo encontrar una aguja en un pajar de 300 dimensiones a encontrarla en un pajar de 3 dimensiones.

Afortunadamente como estamos hablando de una regresión lineal y la función de costo es convexa por lo que esta aguja esta al final del tazón.

Algo muy importante es resaltar que n en la ecuación es nuestro ritmo de aprendizaje, este multiplica al gradiente ya que entre más grande sea, más rápido ira el algoritmo y si es muy rápido, puede que se salte los valores óptimos.

Vamos a crear una implementación de este algoritmo.

ra = 0.1

iteraciones = 1000

m = 100

b = np.random.rand(2,1)

for iteracion in range(iteraciones):

gradientes = 2/m \* x\_b.T.dot(x\_b.dot(b) - y)

b = b -ra \* gradientes

Vamos a explicar parte por parte lo que tenemos en nuestro modelo de gradiente descendente tipo batch:

* Ra esa nuestra métrica de ritmo de aprendizaje, vamos a empezar con un ritmo de 0.1.
* Iteraciones es el número de pasos que queremos que nuestro gradiente realice en nuestro descenso.
* M es el número de datos, como recordamos en nuestro ejemplo de regresión lineal creamos un vector de 100 datos aleatorios.
* B es nuestro vector de parámetros que como sabemos, no importa con cuales convencemos, el algoritmo va a entrenar para mejorarlos.
* En el ciclo for tenemos nuestras iteraciones o pasos, en los cuales, por cada paso, vamos a calcular nuestro vector de gradientes con la formula .
* Por último, en cada paso vamos a calcular nuestro nuevo vector de parámetros b, restando el vector de gradiente.

Una vez corremos el modelo el juego predice el siguiente conjunto de parámetros.

Text

Description automatically generated

Como vemos son los mismos que con la ecuación normal, aunque esta vez utilizamos un algoritmo mucho más amigable computacionalmente hablando. Ya que para un dispositivo es más fácil realizar iteraciones con operaciones de suma, resta, división y multiplicación, a tener que calcular una matriz inversa.

Ahora ¿Qué hubiera pasado si hubiéramos utilizado un ritmo de aprendizaje diferente?

Vamos a graficar los primeros 10 pasos de nuestro modelo, con 3 diferentes ritmos de aprendizaje.

ra = ?

iteraciones = 10

m = 100

b = np.random.rand(2,1)

plt.scatter(x,y,label="Datos")

for iteracion in range(iteraciones):

gradientes = 2/m \* x\_b.T.dot(x\_b.dot(b) - y)

b = b -ra \* gradientes

y\_nuevo = x\_nuevo\_b.dot(b)

plt.plot(x\_nuevo,y\_nuevo,"r-")

plt.show()

Primero con ra = 0.01

Chart, scatter chart

Description automatically generated

Como podemos ver la velocidad es tan lenta que no se ajusta nada a los datos, probablemente en algún momento llegue, pero va un poco lento.

Segundo con nuestra velocidad original de 0.1:

Chart, scatter chart

Description automatically generated

Como podemos ver en este caos en tan solo 10 pasos, el modelo logra ajustarse bastante bien al conjunto de datos.

Por último, una velocidad de 1.

Chart, line chart

Description automatically generated

Como podemos ver el salto es tan grande y tan rápido que se saltó nuestro conjunto de datos y se fue a un punto mucho más alejado del optimo.

Una forma para encontrar una velocidad optima es ver el ritmo de aprendizaje como hiperparámetro y utilizar el método de **grid search** para que este pruebe con diferentes velocidades y te diga cual es la mejor.

Para tener un numero de iteraciones adecuado se puede señalar en el ciclo for una condición para que pare de hacer las iteraciones, por ejemplo, si yo considero que el vector de gradientes es suficiente mente pequeño en 0.0001 se haría de esta manera.

ra = 0.01

iteraciones = 1000

m = 100

b = np.random.rand(2,1)

for iteracion in range(iteraciones):

gradientes = 2/m \* x\_b.T.dot(x\_b.dot(b) - y)

b = b -ra \* gradientes

if gradientes[0] <= 0.0001:

break

Cuando el gradiente es menor o igual a 0.0001 el modelo se detiene.

### **Gradiente de Descenso Estocástico**

El problema principal con el gradiente descendente tipo batch es el hecho de que usa todo el set de entrenamiento para crear gradientes en cada paso o iteración, lo que lo puede hacer algo lento cuando tenemos un set de datos grande.

En este caso el gradiente descendiente estocástico toma un evento aleatorio en nuestro set de entrenamiento en cada paso o iteración y crea gradientes basándose solo en un evento.

Esto obviamente hace que el modelo sea bastante más rápido ya que solo trabaja en un evento por iteración lo que hace que tenga gran cantidad de datos menos que manipular.

Esto también hace posible que podamos trabajar con sets de datos enormes, ya que solo necesita de una sola instancia del set de datos.

Aunque, por otro lado, este modelo por su naturaleza estocástica, que como bien sabemos los modelos estocásticos trabajan con aleatoriedad, este modelo será mucho menos regular que el método batch.

Esto debido a que en vez de descender gentilmente como el tipo batch hasta que llega al mínimo. La función de costo va a dar rebotes arriba y abajo como una pelota, decreciendo solamente en porcentaje, al paso del tiempo este terminara cerca del mínimo e incluso una vez ahí va a seguir dando pequeños rebotes, nunca quedándose estático.

Diagram, engineering drawing

Description automatically generated

En la imagen podemos observar la diferencia entre un descenso de gradiente como el tipo batch y un descenso del tipo estocástico.

Una ventaja bastante interesante del gradiente estocástico es que cuando tienes una función con mínimos globales como esta:

Diagram

Description automatically generated

Todos los rebotes que tiene este tipo de gradiente pueden ayudar a que se salga del mínimo local mucho más rápido y se encuentre el mínimo global.

Por lo tanto, se podría decir que la aleatoriedad es buena para escapar de un óptimo local, aunque esto significa que hay posibilidades de que el modelo nunca se quede en algún mínimo ya que nunca para de moverse.

Una forma utilizada para resolver este dilema es reducir el ritmo de aprendizaje paulatinamente. Por lo que los pasos al inicio serán grandes, ayudando a que se salga de un bache local y estos se irán haciendo más pequeños ayudando al modelo a que se quede estable en el mínimo.

Hay una función que determina a que ritmo se tiene que modificar el ritmo de aprendizaje en cada iteración y se llama horario de aprendizaje.

Esto ya que si el ritmo de aprendizaje se reduce muy rápido puede que el gradiente sé que estancado en un mínimo local, o que se quede estancado a la mitad del camino de cualquier mínimo.

Y si el ritmo de aprendizaje se incrementa muy rápido, puede que encuentre la solución lo bastante rápido y se la salte, dejándote con un valor muy fuera del optimo.

Vamos a crear un modelo de gradiente estocástico con una función de horario de aprendizaje.

epochs = 50

t0 , t1 = 5, 50

m = 100

def horario\_aprendizaje(t):

return t0 / (t + t1)

b = np.random.rand(2,1)

for epoch in range(epochs):

for i in range(m):

random\_index = np.random.randint(m)

xi = x\_b[random\_index:random\_index+1]

yi = y[random\_index:random\_index+1]

gradientes = 2 \* xi.T.dot(xi.dot(b) - yi)

ra = horario\_aprendizaje(epoch \* m + i)

b = b -ra \* gradientes

Antes de explicar lo que es cada parte de esta función hay que explicar que el gradiente estocástico no funciona con un numero de iteraciones especifico como lo hacía el batch.

Este itera conforme a distintos grupos llamados epochs, cada epoch consiste en diferentes iteraciones respecto al número de datos m, por lo que en este caso cada epoch consiste de 100 iteraciones.

Esto se hace porque queremos diferentes ritmos de aprendizaje y la fórmula de este depende del número de epoch en el que se esté iterando.

La función de horario de aprendizaje consiste en 3 variables, la variable t que es la multiplicación de m y el número de epoch más nuestra iteración i. las otras 2 variables t0 y t1 son hiperparámetros que nosotros podemos controlar, y que podríamos optimizar con un grid search por ejemplo.

El modelo consiste en:

* Nuestro número de grupos o epochs.
* Nuestros hiperparámetros t0 y t1.
* Nuestro número de datos m.
* Nuestra función de horario de aprendizaje que comúnmente se define como t0 / (t + t1).
* Un vector aleatorio de parámetros para comenzar.
* En la iteración para solo tomar un valor aleatorio creamos un numero entero aleatorio con la función de numpy rand int, el cual será nuestro índice.
* Con ese índice tomamos un valor aleatorio de nuestro set de datos x, y de nuestra variable a predecir y.
* Ejecutamos nuestra función de gradientes.
* Recalculamos nuestro ritmo de aprendizaje en cada iteración.
* Y calculamos nuestro nuevo vector de parámetros.

Una vez ejecutado nuestro modelo obtenemos estos parámetros:



Y como podemos ver para nuestra función inicial de y = 3 + 3x, la predicción no está nada mal.

Si graficamos los primeros pasos del gradiente estocástico se verían de esta forma.

Chart, line chart

Description automatically generated

Como podemos observar sus pasos no se ven equivalentes hay unos muchos más grandes que otros, pero al final gracias a nuestra función de horario de aprendizaje se reducen bastante ya que se está en los datos.

Para utilizar el gradiente estocástico, scikit también tiene su propio modulo llamado **SGDRegressor**, este módulo funciona optimizando el modelo para la función de costo del error cuadrado o MSE, que es la que hemos utilizado nosotros.

Vamos a ver un ejemplo:

from sklearn.linear\_model import SGDRegressor

sgd = SGDRegressor(max\_iter=1000, tol=1e-3, penalty=None, eta0=0.1)

sgd.fit(x, y.ravel())

Para utilizar el módulo le tienes que indicar como parámetros:

* Max\_iter, el cual es el número máximo de epochs que queremos, en este caso le dijimos que máximo 1000.
* Tol, es la tolerancia por lo que le decimos que si la perdida es menor de 0.001. por lo que en este ejemplo el modelo va a detenerse si llega a los 1000 epochs o si la perdida es menor a esta cantidad.
* Eta, es el ritmo de aprendizaje con el que queremos comenzar, en este caso 0.1.
* En este caso no queremos que el modelo use regularización, es por eso por lo que el parámetro penalty se pone en None, aunque esto de la regularización lo explicaremos un poco más adelante.
* Para entrenar nuestros datos utilizamos nuestro set de entrenamiento x y la variable a predecir y, y estaba en forma de una lista con sublistas, y necesitamos que sea una sola lista, por lo que utilizamos el método ravel.

Text

Description automatically generatedText

Description automatically generated

Una vez tenemos nuestro modelo con los datos entrenados ya solo tenemos que utilizar de nuevo los métodos intercept\_ y coef\_ para ver los resultados.



Como podemos ver los resultados no fueron malos, y la forma de usar el modelo con scikit es bastante sencilla.

### **Descenso de Gradiente Mini-Batch**

El último modelo de gradiente descendente que veremos es el de mini-batch.

Este es bastante fácil de entender una vez que sabemos en qué consisten los tipos batch y estocástico.

En vez de entrenarse utilizando todo el set de entrenamiento como el batch, o utilizando solo una instancia como el estocástico, este calcula gradientes basándose en pequeños sets de datos aleatorios llamados mini-batches o lotes pequeños.

La gran ventaja del mini-batch sobre el estocástico es que puedes obtener un subidón de desempeño desde el punto de vista de optimización de hardware y operaciones con matrices especialmente cuando se utiliza la memoria del dispositivo.

Esto debido a que el parámetro de progreso de este modelo es menos errático que el estocástico, especialmente si se utilizan mini-batches un poco grandes.

En otras palabras, va a ser más fácil para el mini batch acercarse al punto mínimo, aunque va a ser más difícil que salga de un bache local.

Diagram

Description automatically generated

Como podemos observar en la imagen, tenemos una comparación de como cada tipo de descenso de gradiente se acerca al valor mínimo.

Como podemos observar el tipo batch es el único que se detiene en el mínimo mientras que el estocástico y el mini batch se siguen moviendo incluso habiendo encontrado el mínimo.

Aun así, el tipo batch toma más tiempo en cada paso o iteración mientras que él estocástico y el mini batch se mantendrán en el mínimo también si tenemos una buena función de horario de aprendizaje.

Antes de finalizar con el tema de gradiente descendente vamos a ver una tabla donde comparamos todos los métodos que hemos visto para conseguir una regresión lineal, sus ventajas y sus desventajas.

Recordemos que m es el número de instancias que tenemos en nuestro set de datos y n es el número de variables o predictores.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algoritmo | Gran cantidad de instancias m | Puede saturar la memoria computacional | Gran cantidad de variables n | Hiperparámetros | Requiere escalación de variables | Modulo en scikit |
| Ecuación Normal | **Rápido** | **Si** | **Lento** | **No** | **No** | **N/A** |
| Singular value descomposition SVD | **Rápido** | **Si** | **Lento** | **No** | **No** | **LinearRegression** |
| GD Batch | **Lento** | **Si** | **Rápido** | **Si** | **Si** | **SGDRegressor** |
| GD Estocástico | **Rápido** | **No** | **Rápido** | **Si** | **Si** | **SGDRegressor** |
| GD Mini-Batch | **Rápido** | **No** | **Rápido** | **Si** | **Si** | **SGDRegressor** |

Aquí podemos observar que cada uno de nuestros métodos tiene características diferentes, y dependiendo el contexto puedes decidir cual quieres usar, aunque como podemos ver los algoritmos óptimos actualmente si estamos manejando una gran cantidad de datos son los de gradiente estocástico y gradiente mini-batch.

## **Regresión Polinomial**

¿Qué pasa si nuestros datos son mucho más complejos que una tendencia lineal? Pues sorpresivamente se puede utilizar un modelo lineal para adaptarse a datos no lineales.

La forma más simple de hacerlo es añadir potencias a cada variable creando nuevas variables, y luego entrenar el modelo en esta serie de variables con potencias.

En otras palabras, extiende el modelo lineal al agregar predictores o variables adicionales, obtenidos al elevar cada uno de los predictores originales a una potencia.

A esto se le conoce como regresión polinomial.

Por ejemplo, recordemos que una función lineal de una variable se ve así y = mx + b, mientras que una función polinomial de una variable se vería de esta manera y = mx + nx2 + b, donde tenemos 2 parámetros (m y n) en vez de 1 y el bias.

Vamos a practicar este modelo generando un set de datos con 100 instancias que no sea lineal.

Para este ejemplo haremos y = 2 + x + 0.5x2 + ruido con elementos aleatorios de 0 a 1 donde la no linealidad viene principalmente de x2.

m = 100

x = 3 \* np.random.rand(m,1) - 3

y = 2 + x + 0.5 \* x\*\*2 + np.random.rand(m,1)

plt.scatter(x,y)

Chart, scatter chart

Description automatically generated

Como podemos observar un modelo lineal no se adaptaría bien a este set de datos ya que al inicio los datos tienen una tendencia bajista y en un momento esta tendencia se frena y se vuelve alcista.

Una de las formas más sencillas para modelar una regresión polinomial es con el módulo de scikit-learn **PolynomialFeatures**.

Esta clase lo que hace es tomar nuestros predictores o variables y añadirles una potencia, agregándolos como nuevos predictores e intentando de igual manera asignarles un parámetro.

Cabe mencionar que la regresión polinomial nos permite ajustar el grado de potencia que queremos y que más nos ayude a ajustarnos a los datos.

Por ejemplo, una función de segundo grado seria esta y = mx + nx2 + b, pero qué tal si una función con este grado no se ajusta bien a los datos. pues también podemos probar con una función de grado 3, la cual se vería de la siguiente manera y = mx + nx2 + ox3 + b.

Hay que notar que entre más elevamos el grado de la función, más variables estamos creando, al utilizar el grado 3, pasamos de asignarle 1 variable al modelo a tener 3 variables.

Esta es una de las desventajas de la regresión polinomial ya que, si tenemos un gran número de variables y necesitamos un grado elevado, la ecuación se puede volver muy difícil de computar y sobrecargaría la memoria de nuestro dispositivo.

Para nuestro ejemplo vamos a utilizar un grado 2 y ver como se ajusta a nuestros datos.

También por efectos prácticos vamos a pedirle a nuestro modelo que no incluya el bias, ya que solo queremos que nuestro modelo ajuste los predictores o variables.

from sklearn.preprocessing import PolynomialFeatures

poli = PolynomialFeatures(degree=2, include\_bias=False)

Una vez teniendo nuestro modelo ajustado con las características que deseamos vamos a entrenar y transformar nuestro set de datos x.

x\_poli = poli.fit\_transform(x)

Si comparamos nuestra variable x con la nueva variable x\_poli, vamos a notar que ya no solo tenemos un valor, si no que tenemos 2, uno para nuestra variable original y otro de nuestra variable al cuadrado.

x[0], x\_poli[0]



Ahora que nuestro modelo tiene una forma polinomial, vamos a efectuar una regresión con el módulo de scikit LinealRegression.

reg\_lin = LinearRegression()

reg\_lin.fit(x\_poli,y)

reg\_lin.intercept\_,reg\_lin.coef\_



Al pedirle los parámetros a nuestra regresión nos damos cuenta de que la función de predicciones es y = 2.46 + 1.06x + 0.52x2, lo cual no está nada mal tomando en cuenta que nuestra función original sin ruido era y = 2 + x + 0.5x2.

Vamos a ver gráficamente como es que se ajustó la regresión.

x\_nuevo = np.linspace(-3.2,0.2,100)

y\_nuevo = 2.46 + 1.03 \* x\_nuevo + 0.53 \* x\_nuevo \*\* 2

plt.plot(x\_nuevo,y\_nuevo,"r-",label="Predicción")

plt.scatter(x,y,label="Datos")

plt.legend()

plt.show()

Chart, scatter chart

Description automatically generated

De esta manera es como nuestra regresión polinomial se adaptó a las múltiples tendencias de los datos.

Algo importante sobre la regresión polinomial es que, si se tienen distintas variables, esta ayuda a encontrar similitudes entre las variables, cosa que no sucede con la regresión lineal.

Esto debido a que como lo hace el módulo de scikit, si tenemos 2 variables, estas se agregan a la misma ecuación, ejemplo, xy + x2y + xy2.

## **Curvas De Aprendizaje**

Si se sube el grado en el que se utiliza la regresión polinomial, probablemente este se adapte mejor a los datos.

Aunque aquí es donde retomamos el tema de overfitting y underfitting, ya que, en este tipo de regresión, si el grado que utilizas es muy bajo puede que esta no se adapte bien a la tendencia de los datos, pero si el grado que se utiliza es demasiado alto, este va a intentar adaptarse tanto a los datos que cometerá overfitting y no hará buenas predicciones.

Graphical user interface

Description automatically generated

En la imagen se puede observar cómo se utilizó un grado 1 para los datos y este no se adaptó nada bien a ellos, mientras que utilizando el grado 4 la función se adaptó casi perfecto a la función original, por último, con un grado 15, se cometió tanto overfitting que en momentos la función no seguía la tendencia de los datos.

¿Cómo sabemos que grado elegir?

En el capítulo 2 vimos una introducción a lo que es la validación cruzada o cross validation para obtener una medida de desempeño en nuestro modelo. Si el modelo se desempeña bien en nuestro set de entrenamiento, pero al crear diferentes folds no se desempeña bien, entonces hay una muestra de overfitting.

Si tanto en el set de entrenamiento como en los folds de validación cruzada, no hay un buen desempeño, entonces probablemente haya underfitting.

De esta manera podemos darnos una idea de si tenemos que subir o bajar el grado polinomial.

Otra forma de darnos una idea de en qué grado tendríamos un buen desempeño es con el método llamado curvas de aprendizaje.

Las curvas de aprendizaje es una gráfica donde vamos a ver el desempeño de nuestro modelo según una función de costo como RMSE para diferentes eventos o instancias.

Lo común es que se haga una gráfica tanto para el set de entrenamiento, como para el set de validación.

Para generar las curvas de aprendizaje tenemos que entrenar los datos en diferentes subsets del set de entrenamiento, cada uno con un tamaño diferente.

Vamos a crear una función que nos ayude a graficar las curvas de aprendizaje.

from sklearn.metrics import mean\_squared\_error

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

def curvas\_aprendizaje(modelo ,x , y):

x\_train, x\_val, y\_train, y\_val = train\_test\_split(x, y, test\_size=0.2)

train\_error, val\_errors= [], []

for m in range(1,len(x\_train)):

modelo.fit(x\_train[:m], y\_train[:m])

y\_train\_pred = modelo.predict(x\_train[:m])

y\_val\_predict = modelo.predict(x\_val)

train\_error.append(mean\_squared\_error(y\_train[:m],y\_train\_pred))

val\_errors.append(mean\_squared\_error(y\_val,y\_val\_predict))

plt.plot(np.sqrt(train\_error), "r-+", label = "set de entranamiento")

plt.plot(np.sqrt(val\_errors), "b-", label = "set de validación")

plt.legend()

Vamos a repasar paso a paso lo que estamos haciendo con esta función:

* Nuestros parámetros van a ser, el modelo que deseamos usar (LinearRegression, SGDRegressor, etc.), nuestros datos para la variable x que son los predictores, y los datos de la variable y, que es la variable por predecir.
* Empezamos la función creando un set de entrenamiento y una réplica llamada set de validación tanto para x como para y con 80% para entrenamiento y 20% para prueba.
* Creamos 2 listas vacías que es donde vamos a guardar el error RMSE que vayamos calculando a lo largo de la iteración.
* Después iteramos desde el valor 1, a el valor total de datos de nuestro set de entrenamiento que en este caos es 80, por lo que deberíamos tener 80 iteraciones.
* En cada iteración vamos a entrenar nuestros datos con el modelo que seleccionamos, pero en el set de entrenamiento, solo vamos a entrenar con instancias de acuerdo con el valor de la iteración en el que vayamos, por ejemplo, en la primera iteración solo vamos a entrenar un solo dato o evento en todo el set, mientras que en la iteración 50, vamos a entrenar 50 datos.
* Para el set de validación entrenaremos siempre todo el set completo.
* Generamos una predicción tanto con el set de entrenamiento, y con el set de validación, y para cada predicción calculamos la raíz cuadrada del MSE, que es igual al RMSE.
* Una vez teniendo todos los errores generados, los graficamos.

Ahora que tenemos nuestra función vamos a probarla con una regresión lineal, o un grado 1 (una simple línea).

De antemano sabemos que los datos se ven de esta manera.

Chart, scatter chart

Description automatically generated

Por lo que ya sabemos que un grado uno no tendrá un buen desempeño, pero vamos a ver como se ven sus curvas de aprendizaje.

reg\_lin = LinearRegression()

curvas\_aprendizaje(reg\_lin, x, y)

Chart, line chart, histogram

Description automatically generated

Como vemos en las primeras instancias de nuestro set de entrenamiento, tenemos un error de casi 0, esto porque adaptarse a un solo dato no es nada difícil para ningún modelo.

Es a partir de los 10 datos que el modelo empieza a ser errático, ya que como vemos en la gráfica de los datos, estos tienen diferentes tendencias.

De hecho, esta grafica de curvas de aprendizaje tiene la típica forma de un modelo que tiene underfitting.

Vamos a probar lo mismo, pero con un modelo polinomial de grado 10.

Para esto vamos a crear un pipeline, que agregue las características de la regresión polinomial, y luego genere las predicciones con el módulo LinearRegressor.

from sklearn.pipeline import Pipeline

regresion\_polinomial = Pipeline([

("carac\_polinomiales", PolynomialFeatures(degree=10, include\_bias=False)),

("reg\_lin", LinearRegression())

])

Una vez tenemos nuestro pipeline que genera regresiones polinomiales, vamos a generar las curvas de aprendizaje.

curvas\_aprendizaje(regresion\_polinomial,x,y)

Chart, line chart

Description automatically generated

Lo primero que hay que notar en este caso es que, cuando teníamos un solo grado, el error convergía en aproximadamente 0.5, en este caso el error converge en alrededor de 0.25, por lo que podemos asumir que este modelo se adapta mejor.

El problema es que aún existe una gran separación entre el set de entrenamiento y el set de validación, por lo que hay señales de overfitting.

Si cambiamos el grado de polinomio de nuestro pipeline a 2, y graficamos las curvas de aprendizaje:

regresion\_polinomial = Pipeline([

("carac\_polinomiales", PolynomialFeatures(degree=2, include\_bias=False)),

("reg\_lin", LinearRegression())

])

Chart

Description automatically generated

Como vemos la separación entre ambos sets disminuyo bastante, y los 2 se desempeñan casi igual de bien, por lo que, entre estas 3 opciones, la del grado numero 2 es la mejor.

## **Modelos Lineales Regularizados**

Como vimos en el capítulo 1 y 2, una forma de reducir el overfitting en un modelo es regularizarlo (por ejemplo, restringirlo), mientras menos grados de libertad este tenga, más complicado será que este produzca overfitting.

Una de las maneras más simples de regularizar un modelo polinomial es reduciendo sus grados polinomiales.

Para un modelo lineal, la regularización normalmente se consigue al restringir los pesos del modelo.

Aquí es donde aprenderemos en qué consisten la regresión Ridge, la regresión Lasso y la red elástica, los cuales son modelos que tienen diferentes métodos para restringir los pesos.

### **Regresión Ridge**

La regresión Ridge también conocida como la regularización Tikhono, es una versión regularizada de una regresión lineal.

En este caso a la ecuación le añadimos un término de regularización a la función de costo, en este caso MSE.

Text

Description automatically generated with medium confidence

Esto es lo que le permite al algoritmo, no solo adaptarse a los datos, sino que también reduce los pesos a lo más pequeño posible.

Es importante mencionar que este término solo se añade a la ecuación durante el entrenamiento de datos, una vez que el modelo este entrenado, hay que usar la función de costo normal para medir el error.

El símbolo es el que delimita que tanto se va a regularizar la función.

Chart, scatter chart

Description automatically generated

A lo que nos referimos con regularización, es a lo siguiente.

Como se puede ver en la imagen, los puntos verdes simbolizan al set de entrenamiento y los puntos rojos simbolizan al set de prueba.

En este caso queremos realizar una regresión para el set de prueba, con los métodos que vimos antes como la función normal, este intentara ajustarse lo mejor posible a los datos que tiene, por lo que los va a cruzar perfectamente como la línea roja, y el error será 0.

Esto sería perfecto si el modelo solo se fuera a usar con los datos del set de prueba, la cosa es que al momento de usar el modelo con el set de entrenamiento esta línea va a estar bastante alejada de los datos incrementando el error.

Es aquí donde entra la regresión Ridge y el parámetro , que es el que añade el peso, al usarlo este inclina la línea, que, aunque si solo la medimos con el set de prueba tendría un poco de error, al medirla con el set de entrenamiento, va a tener muchos mejores resultados, reduciendo por mucho el overfitting.

Si = 0, entonces tendremos los mismos resultados que con una regresión lineal normal.

Y si por ejemplo le añadimos mucho peso con un muy alto, entonces la línea será prácticamente horizontal sin inclinación alguna, y probablemente no tenga muy buenos resultados, es por eso por lo que hay que controlar cuanto peso le asignamos.

Matemáticamente la ecuación para hacer la regresión Ridge es:

Logo

Description automatically generated with medium confidence

* Lo primero que hacemos es hacer una regresión lineal normal.
* MSE es la función de costo para saber el error que tenemos, en este caso ya sabemos que se puede calcular con la función de scikit mean\_squared\_error.
* Una vez tenemos el MSE de la regresión calculado, vamos a elevar nuestro vector de parámetros b al cuadrado y lo vamos a multiplicar por ½ y por el valor que le asignemos a .

Vamos a utilizar el ejemplo pasado donde calculamos una regresión lineal.

Tenemos estos datos:

Chart, scatter chart

Description automatically generated

Al hacer la regresión lineal con LinearRegressor, tenemos los siguientes parametros.



Por lo que y = 3.46 + 3.08x.

Después tenemos que calcular el MSE.



Una vez que tenemos nuestro MSE calculado, hay que elevar nuestro vector de parámetros al cuadrado, OJO sin incluir el bias. Por lo que solo elevaremos 3.08.

Supongamos que asignamos un = 0.5.

Nos quedaría la ecuación Ridge = 0.76 + 0.5 \* ½ \* 3.082.



Por lo que nuestra ecuación con regresión Ridge quedaría:

Y = 3.46 + 4.81x

Vamos a graficar nuestras 2 regresiones:

Chart, scatter chart

Description automatically generated

Como podemos ver al agregar un alfa de 0.5, nuestra regresión tiene más peso y se inclina hacia abajo.

SIEMPRE es necesario escalar los valores de los datos cuando usamos este tipo de regresión ya que, es bastante sensible a las escalas.

También se puede hacer ridge regresión con regresiones polinomiales, este es un ejemplo de cómo se vería.

Chart

Description automatically generated

Una forma más simple de realizar una regresión Ridge es con el módulo de scikit, Ridge.

Este módulo fue optimizado de manera que realiza el cálculo de parametros por medio de matrices y con un método de factorización creado por Andre-Louis Cholesky, pero que básicamente llega al mismo resultado que con la fórmula que acabamos de ver.

Supongamos que tenemos una regresión polinomial de grado 2, y = 3 + 4x + 2.5x2.

Para obtener su equivalente con la regresión ridge:

Primero escalamos nuestras variables *x* y *y*.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x = scaler.fit\_transform(x)

y = scaler.fit\_transform(y)

Después importamos Ridge y como parámetros seleccionamos que queremos un alfa de 0.5.

from sklearn.linear\_model import Ridge

ridge = Ridge(alpha=1)

ridge.fit(x,y)

ridge.predict([[4],[2.5]])



Así de sencillo ahora tenemos una regresión ridge que se define de la siguiente manera:

Y = 3 + 3.96x + 2.47x2

Hay que recordar que este tipo de regresión se usa principalmente para eliminar overfitting, por lo que normalmente solo se usa una vez teniendo tu regresión lineal o polinomial regulares y habiendo confirmado que tienen overfitting.

Después de eso puedes buscar algún valor para alfa que te ayude a reducir ese overfitting.

Otra forma de realizar Ridge es con el parámetro penalty, este es un hiperparámetro que de igual manera que alfa, ajusta la regresión agregándole peso, para reducir el overfitting.

Penalty tiene 3 posibilidades:

* L2 que es la regresión ridge
* L1 que es la regresión Lasso
* Elasticnet que es una red elástica.

La podemos usar por ejemplo con la función SGDRegressor:

sgd = SGDRegressor(penalty="l2")

sgd.fit(x,y.ravel())

sgd.predict([[4],[2.5]])



### **Regresión Lasso**

Lasso viene de least absolute shrinkage and selection operator, o en español operador de selección y contracción mínima absoluta. Convenientemente conocido como Lasso.

Es otro método para regularizar una regresión lineal.

Justo como la regresión Ridge, este agrega un parámetro , pero en este caso utiliza la norma L1 de vector en vez de la norma L2 de Ridge que era multiplicar el cuadrado de cada parámetro por un medio.

La norma L1 consiste únicamente en el valor absoluto del vector de parametros, por lo que la ecuación de parámetros para la regresión Lasso seria:



Lo que significa que para ejecutarla tenemos que seguir exactamente los mismos pasos que con la regresión Ridge, simplemente que en vez de elevar al cuadrado el vector de parámetros y multiplicarlo por 1/2, calculamos el valor absoluto del vector y después lo multiplicamos por nuestro alfa.

Chart, scatter chart

Description automatically generated

Realmente el funcionamiento de Lasso y Ridge es bastante similar, aunque hay ciertas diferencias importantes que te podrían ayudar a decidir cuál de los métodos utilizar en un cierto contexto.

Una de las características más importantes de Lasso es que tiende a eliminar los pesos de las variables menos importantes, por ejemplo, dejarlas en 0.

Chart

Description automatically generated

En esta imagen podemos observar que con alfa en 0.1 el modelo pasa a ser prácticamente cubico o de grado 3 x3. Esto sucedió porque el modelo detecto que había variables o instancia que no tenían peso o importancia para la predicción y las elimino, por lo que claramente se puede ver una reducción de saltos y overfitting haciendo el modelo más fluido y adaptable para un set nuevo de datos.

Entonces la diferencia principal entre Lasso y Ridge es que Lasso puede excluir estas variables sin importancia.

Ejemplo:

Diagram

Description automatically generated

Tenemos estos 2 modelos para predecir quien tiene diabetes, en el verde las variables o predictores son bastante importantes ya que tanto la alimentación como la genética, los hábitos alimenticios y si hace deporte o no influyen en personas que tienen diabetes.

Mientras que, en el modelo azul, realmente solo la variable de alimentación tiene importancia, las otras variables realmente no sirven para diferenciar entre alguien que tiene diabetes o no.

En este caso para evitar overfitting sería más recomendable usar Lasso en el modelo azul ya que este nos va a ayudar a reducir peso de las variables que no sirven.

Para modelo verde sería recomendable utilizar Ridge ya que como todas las variables importan, se mantendrá un balance entre ellas.

Para usar Lasso de una manera sencilla, scikit tiene el módulo llamado Lasso.

Este funciona igual que el de Ridge, ejemplo:

from sklearn.linear\_model import Lasso

lasso = Lasso(alpha=0.1)

lasso.fit(x,y)

lasso.predict([[4],[2.5]])



Como vemos es bastante sencillo de utilizar, y puede ayudarnos a reducir la posibilidad de overfitting.

### **Red Elástica**

La red elástica es otro ejemplo de una regresión lineal regularizada y podríamos decir que es como el punto medio entre Ridge y Lasso.

El método consiste en una mezcla entre Lasso y Ridge, donde añadimos un parámetro de mezcla r.

Este parámetro r va de 0 a 1 y si r =0 significa que la regresión será tipo Ridge, mientras que si el parámetro es r = 1, entonces la regresión será exactamente igual al tipo Lasso.

Por lo que podemos asumir que r=0.5 es una combinación 50% Ridge, 50% Lasso.

La fórmula matemática se ve de la siguiente manera:



Como podemos observar, esta fórmula es prácticamente la misma, a excepción que añadimos r de ambos lados haciendo que la fórmula de Ridge vaya al contrario con 1-r, y sumamos el error, el cálculo Lasso y el cálculo Ridge.

Te preguntaras ¿Cuándo se recomienda usar la red elástica?

Pues la respuesta es sencilla, se recomienda usar la red elástica cuando no tienes conocimiento de la importancia de tus variables y no estás seguro de si debes utilizar Lasso o Ridge.

Siempre es recomendable tener un poco de regularización en el modelo, por lo que una forma de empezar es aplicando la red elástica a tus modelos de regresión lineal.

Para utilizarlo vamos a usar el módulo de scikit ElasticNet donde además de ingresar el parámetro alfa, tenemos que indicar cual va a ser el parámetro r, l1\_ratio.

from sklearn.linear\_model import ElasticNet

net = ElasticNet(alpha=0.1, l1\_ratio=0.5)

net.fit(x,y)

net.predict([[4],[2.5]])



Y así de sencillo es, ahora tenemos un modelo que estima parámetros con 50% regularización Ridge y 50% regularización Lasso.

### **Early Stopping**

Early stopping o literalmente en español detención prematura, es una forma de optimizar algoritmos que funcionan iterativamente como el descenso de gradiente.

Recuerdas que, por ejemplo, para el descenso de gradiente estocástico creábamos diferentes pasos o iteraciones, aunque este no se detenía incluso llegando al mínimo, por lo que tuvimos que introducir una función de horario de aprendizaje.

Chart, line chart

Description automatically generated

Pues hay un método para hacer que este se detenga cuando llega al mínimo, y se llama early stopping.

A picture containing chart

Description automatically generated

Como podemos ver, al entrenar diferentes sets de datos, el hecho de que el gradiente nunca se detenga, puede hacer que los parámetros se salgan del valor optimo y se dirijan a otro lado, creando separación y contribuyendo al overfitting.

Exactamente para evitar esto es que se implementó el early stopping, para que podamos hacer que nuestro algoritmo detecte cuando ya está en el valor optimo y se detenga en esa posición.

Vamos a crear un ejemplo de early stopping para el gradiente estocástico.

Para empezar, tenemos que crear nuestro set de entrenamiento y nuestro set de validación.

Como sabemos para ejecutar un gradiente estocástico los datos siempre tienen que estar escalados.

Y para crear overfitting intencionalmente y compararlo después vamos a convertirlo a un polinomio de grado 90.

Para eso vamos a crear un pipeline.

x\_train, x\_val, y\_train, y\_val = train\_test\_split(x, y, test\_size=0.2)

escalar\_poli = Pipeline([

("carac\_polinomiales", PolynomialFeatures(degree=90, include\_bias=False)),

("escalar", StandardScaler())

])

x\_train\_tr = escalar\_poli.fit\_transform(x\_train)

x\_val\_tr = escalar\_poli.fit\_transform(x\_val)

Después de esto vamos a crear nuestro modelo SGD, vamos a asegurarnos de que el ritmo de aprendizaje sea constante y sea de 0.0005.

sgd = SGDRegressor(max\_iter=1,tol=-np.infty,warm\_start=True, penalty=None, learning\_rate="constant", eta0=0.0005)

En este caso agregamos los parámetros de max:\_iter, tol y warm\_start, porque como vamos a estar entrenando los datos en un bucle for, queremos que no empiece a entrenar desde 0 en cada iteración si no que continue con su camino especifico.

Una vez teniendo nuestro modelo y nuestros datos listos, creamos nuestro early stoping.

from copy import deepcopy

minimum\_val\_error = float("inf")

best\_epoch = None

best\_model = None

val = []

for epoch in range(1001):

sgd.fit(x\_train\_tr,y\_train.ravel())

y\_val\_pred = sgd.predict(x\_val\_tr)

val\_error = mean\_squared\_error(y\_val, y\_val\_pred)

val.append(val\_error)

if val\_error < minimum\_val\_error:

minimum\_val\_error = val\_error

best\_epoch = epoch

best\_model = deepcopy(sgd)

plt.plot(val)

Para crear una copia del modelo que mejor se adaptó a los datos necesitamos importar la función deepcopy, la cual va a generar un modelo SGD al cual le indiquemos que copie.

Necesitamos crear las variables vacías que queremos calcular en las iteraciones por lo que empezamos con el valor mínimo, este va a ser el responsable de saber sonde está el mejor modelo por lo que indicamos que es infinito, para estar seguros de que no le estamos dando un valor con el que podemos confundir nuestra iteración.

También queremos saber en que epoch o paso esta nuestro mejor modelo y por último queremos guardar el modelo que tenga los mejores resultados.

Empezamos la iteración con 1000 pasos y primero vamos a entrenar los datos con nuestra variable x o set de entrenamiento.

Al tener el modelo entrenado este se adaptó solo al set de entrenamiento, pero queremos ver si desempeña bien con el set de validación, por lo que hacemos las predicciones con el set de validación.

Con estas predicciones medimos el MSE, y creamos un condicional especificando que cuando se encuentre un valor de error mayor al de la iteración pasada, nos guarde en que paso estuvo el mejor modelo y nos cree una copia de ese modelo para utilizarlo después.

Chart

Description automatically generated

Al correr el modelo ahora podemos graficar los errores en los 1000 epochs.

Se puede observar claramente que alcanza un valor más optimo entre más se acerca al final, vamos a ver en que paso tuvo un mejor desempeño.

best\_epoch



Esta en el paso 955 de 1000, para saber que error tiene podemos hacer esto.

val[955]



En el paso 955 solo tiene un error de 0.097, por lo que no está nada mal.

Y ahora para hacer predicciones con ese modelo solo necesitamos usar nuestra variable **best\_model**.

best\_model.predict(x,y)

## **Regresión Logística**

Como vimos en el capítulo 1, algunos modelos de regresión se pueden usar para clasificación y viceversa.

La regresión logística se usa normalmente para estimar la probabilidad de que un evento o instancia pertenezca a cierta clase (clasificación).

Si la probabilidad estimada es mayor a 50% entonces el modelo la clasifica como que, si pertenece a esa clase, por ejemplo, queremos saber si un objeto es rojo y el modelo lo capta como 70% rojo, entonces lo clasifica como rojo.

Por esto mismo funciona para saber si cierta instancia es de una clase o no es de esa clase, por lo que es un clasificador binario.

### **Estimar Probabilidades**

¿Como funciona la regresión logística?

Como una regresión lineal, una regresión logística no regresa un modelo para hacer predicciones con diferentes valores, si no que te regresa un modelo donde intenta clasificar si un objeto pertenece a cierta instancia o no.

La forma matemática de ver cómo funciona esta regresión es la siguiente:



Donde es la probabilidad de que nuestro objeto u evento pertenezca a cierta clase o no.

b es nuestro vector de parámetros y x nuestro vector de valores ya conocidos, justo como en la regresión lineal.

La gran diferencia la hace L, también conocida como ecuación logística y gracias a la que se le debe el nombre a este tipo de regresión.



A la ecuación logística se le conoce también como función sigmoide ya que regresa un valor que esta SIEMPRE entre el 0 y el 1.

Chart, line chart

Description automatically generated

Este tipo de ecuación es perfecta para modelar nuestro modelo ya que al usarla cuando introduzcamos nuevos datos, estos tendrán un valor asignado de 0 a 1, indicando la probabilidad de que pertenezcan la clase que queremos predecir.

A picture containing shape

Description automatically generated

Lo más común, es encontrar que si la probabilidad es mayor a 50% el modelo lo clasifique como un 1, que significa que si pertenece a clase.

Si la probabilidad es menor al 50% entonces el modelo lo regresa como un 0, que significa que este no pertenece a la clase.

### **Entrenamiento y Función de Costo**

Una vez que sabemos cómo funciona la regresión logística tenemos que saber cómo se entrenan los datos.

El objetivo principal al entrenar los datos es modelar nuestro vector de parámetros b, para que este regrese probabilidades altas mientras que la instancia cumpla con las características que queremos detectar y que regrese probabilidades bajas si la instancia no tiene estas características.

Para lograr esto utilizamos una función de costo, el modelo aprende si una instancia es 1 o 0, mientras el costo va aumentando o disminuyendo conforme a la probabilidad de que el objeto sea 0 o 1 respectivamente.

Para lograr esto se utiliza una función logarítmica multiplicada por -1:

A picture containing text, clipart

Description automatically generated

Chart, line chart

Description automatically generated

Cuando se tiene un objeto que pertenece a la clase que queremos predecir, tenemos la función de costo de color azul c = -log(p), donde c es el costo y p es la probabilidad de que este objeto sea de la clase 1, por lo que en esta categoría entran las instancias que resultaron con una probabilidad de más de 50%, y como vemos el costo es de 0 cuando la probabilidad es de 100% y cuando la probabilidad más se acerca a 0 el costo va creciendo exponencialmente.

Cuando el modelo clasifica una instancia como que no pertenece o 0, la función de costo será la línea verde, c = log(1-p), por lo que cuando la probabilidad sea de 100% de que no pertenece a la clase, el costo será de 0, pero igualmente mientras más se aleje del 0, el costo de esta ira creciendo.

Entonces, para determinar el costo dentro de todo nuestro set de entrenamiento, utilizamos la suma ponderada para cada una de nuestras instancias, divididas entre el número de instancias.



Puede parecer confuso que tengamos las 2 operaciones antes vistas en la misma ecuación, pero esta está hecha de manera que si y = 0, la primera parte de la ecuación  se elimina, mientras que sí y = 1,  esta parte se elimina, dejando solo el cálculo de función de costo que vimos anteriormente.

Ahora que sabemos cómo se calcula el costo de una regresión logística, tenemos que saber cómo hacer para encontrar el set de parámetros b que encuentre el costo mínimo de la función.

Al ser una función convexa significa que podemos usar el método de derivadas parciales para saber cómo afecta el cambio del vector de parámetros b, conforme la función de costo , para lo que podemos resolver como con la regresión lineal, usando el descenso de gradiente, hasta que este encuentre el valor de parámetros, que nos del costo mínimo.

### **Fronteras De Decisión**

Vamos a ver un ejemplo de regresión lineal.

Para esto vamos a crear un dataset de perfiles de candidatos de empleados, que contenga su resultado en el examen de matemáticas GMAT, su promedio de calificaciones GPA y sus años de experiencia laboral, así como un parámetro con 1 si un empleado fue admitido, y 0 si n fue admitido.

import pandas as pd

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

Table

Description automatically generated

Lo que queremos hacer es crear un modelo de clasificación que basándose en las variables gmat, gpa y experiencia de trabajo nos genere una predicción de si este será admitido o no.

Para crear el modelo de regresión utilizaremos el módulo de scikit, **LogisticRegression**, pero antes de proporcionarle datos vamos a separar nuestras variables predictoras, de nuestra variable a predecir.

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

y = df['admitted']

Creamos nuestro modelo y generamos predicciones:

from sklearn.linear\_model import LogisticRegression

clf = LogisticRegression()

clf.fit(x,y)

y\_pred = clf.predict(x)

Después, vamos a utilizar una función de este modelo que se llama **predict\_proba**, el cual nos va a regresar una lista con la probabilidad de que, si sea admitido, y la probabilidad de que no sea admitido.

y\_proba = clf.predict\_proba(x)

Text

Description automatically generated

Y vamos a crear una sola lista con las probabilidades de que si pertenezca y otra con las probabilidades de que no pertenezca.

y\_proba1 = [y\_proba[i][1] for i in range(0,len(y\_proba))]

y\_proba2 = [y\_proba[i][0] for i in range(0,len(y\_proba))]

Una vez tenemos nuestras predicciones creadas, y nuestras listas de probabilidad vamos a añadirlas a nuestro dataframe.

df["predict"] = y\_pred

df["proba1"] = y\_proba1

df["proba2"] = y\_proba2

df = df.sort\_values(by=["work\_experience"])

df.head()

Table

Description automatically generated

Ahora vamos a ver el rango de decisión por ejemplo si lo compramos con los años de experiencia.

plt.plot(x,df["proba1"],"b--")

plt.plot(x,df["proba2"],"r--")

plt.scatter(x,[df["predict"]==1])

plt.scatter(x,[df["predict"]==0])

Chart, line chart, scatter chart

Description automatically generated

Como podemos ver el color azul indica que el trabajador si será admitido y el color rojo indica que el trabajador no será admitido.

Es claro que la gente que tiene de 4 a 6 años de experiencia tiene muchas más probabilidades de ser admitido que los que tienen de 1 a 3 años de experiencia.

Por ejemplo, en nuestro dataframe, todos los candidatos que tienen 5 o 6 años de experiencia fueron admitidos.

Table

Description automatically generated

Para el modelo donde le cuesta trabajo decidir es en los candidatos que tienen entre 3 y 4 años de experiencia, que es donde se cruzan nuestras líneas de probabilidades.

Table

Description automatically generated

Aquí claramente se puede ver que, aunque los que tienen 4 años de experiencia tienen una probabilidad más al aplicar, hay ciertos candidatos que nos son seleccionados con 4 años, y ciertos candidatos que si lo son.

En esto pueden inferir también las otras variables, aunque lo interesante aquí es que se justo como vimos en nuestro proyecto de clasificación, aquí también se puede regularizar el umbral de selección, solo que en este caso utilizaríamos las mismas normas que con las regresiones, como L1 y L2.

Chart, scatter chart

Description automatically generated

Con la ayuda de la regularización, podemos agregarle peso a esa línea de decisión y flexibilizar el modelo tanto para admitir a más personas o para rechazar a más candidatos, dependiendo de las necesidades que tenga la empresa.

Lo único que tenemos que hacer para regularizar una función logística con scikit, es agregar el parámetro al momento de crearla.

Ejemplo:

LogisticRegression(penalty=”elasticnet”)

### **Regresión SoftMax**

La regresión logística también se puede modelar de forma múltiple, esto significa como vimos antes, que no tiene que ser binaria, puede clasificar diferentes clases en un solo modelo.

Al crear una regresión logística de forma múltiple se le conoce como regresión softmax.

Este tipo de clasificador funciona creando un set de diferentes puntajes, donde evalúa cada una de las instancias de nuestros datos, para cada una de las clases que queremos predecir.

Por ejemplo, supongamos que tenemos 3 clases, A B C, pues entonces el modelo creara un puntaje para cada clase haciendo esta operación. Donde evaluara la instancia x.

Text, letter

Description automatically generated

Donde b es el vector de parámetros asignado para cada una de las clases, generado por nuestro set de entrenamiento.

Una vez tenemos el puntaje de cada clase podemos calcular la probabilidad de que cada instancia pertenezca a cada clase.

Text

Description automatically generated

Donde L es la función logística o sigmoide que hará que nuestro resultado este siempre entre 0 y 1, asegurándonos que es una probabilidad.

Text

Description automatically generated with low confidence

Asegurándonos esta forma:

Chart

Description automatically generated

Una vez tenemos nuestras probabilidades calculadas, la regresión softmax funciona como los otros clasificadores múltiples, selecciona el puntaje más alto, y lo clasifica como esa clase.

Para realizar una regresión softmax con scikit, lo único que debemos hacer es agregar el parámetro multinomial, al módulo de **LogisticRegression**.

Ejemplo:

from sklearn.linear\_model import LogisticRegression

clf = LogisticRegression(multi\_class="multinomial")

clf.fit(x,y)

y\_pred = clf.predict(x)

Si se requiere regularizar, se puede usar el parámetro penalty de igual manera que con una regresión logística.

Chart, scatter chart

Description automatically generated

Así es como se vería una regresión softmax, por ejemplo, con las clases A, B, C. donde se asignan 3 diferentes espacios para 3 diferentes tipos de clase y cada espacio es una frontera de decisión.