Índice

Contenido

Aprendizaje automático (AA)

Los sistemas de AA *aprenden* cómo combinar entradas para generar predicciones útiles sobre datos nunca vistos. Entrenar un modelo significa aprender (determinar) valores correctos para todos los ejemplos(x) y las ordenadas al origen (y) de los ejemplos etiquetados. En un aprendizaje supervisado, un algoritmo de aprendizaje automático construye un modelo al examinar varios ejemplos e intentar encontrar un modelo que minimice la pérdida. Este proceso se denomina minimización del riesgo empírico.

Notación

Revisemos alguna notación usada en este documento

Atributo (o característica) es una variable de entrada
x1,x2,x3,...,xn

Ejemplo es una instancia de datos en particular
x se coloca en negrita para indicar que es un vector

los ejemplos se dividen en:

- **etiquetados** y
- **no etiquetados**

Los ejemplos etiquetados son ejemplos de los cuales sabes la respuesta, es decir dado un ***x*** conoce el valor ***y*** asociado, los ejemplos no etiquetados son aquellos de los cuales desconoces la respuesta, es decir dado un ***x*** desconocemos su valor ***y*** asociado.

Notación para Ejemplos etiquetados {características, etiqueta} o (x,y)
labeled examples: {features, label} or (x,y)

Los ejemplos etiquetados se usan para entrenar el modelo.

Notación para Ejemplos no etiquetados
unlabeled examples: {características, ?} o (x,?)

Modelo. Define la relación entre los atributos y la etiqueta.

Por ejemplo, un modelo de detección de spam podría asociar de manera muy definida determinados atributos con "es spam". Que el correo tenga un origen específico o que contiene la palabra: ‘ganaste’, o contenga la palabra: ‘gran promoción´ podría indicar que es spam.

Destaquemos dos fases en el ciclo de un modelo:

Entrenamiento significa crear o enseñar al modelo. Es decir, le muestras ejemplos etiquetados al modelo y permites que este aprenda gradualmente las relaciones entre los atributos y la etiqueta.

Inferencia significa aplicar el modelo entrenado a ejemplos sin etiqueta. Es decir, usas el modelo entrenado para realizar predicciones útiles (y'). Por ejemplo, durante la inferencia, puedes predecir valores para nuevos ejemplos sin etiqueta.

Observe que un modelo se usa para predecir (inferir) valores (etiquetas y) desconocidos para ejemplos (x) dados. Estas predicciones pueden ser para valores continuos o discretos.

Regresión frente a clasificación

Un modelo de **regresión** predice valores continuos. Por ejemplo, los modelos de regresión hacen predicciones que responden a preguntas como las siguientes:

¿Cuál es el valor de una casa en California?

¿Cuál es la probabilidad de que un usuario haga clic en este anuncio?

Un modelo de **clasificación** predice valores discretos. Por ejemplo, los modelos de clasificación hacen predicciones que responden a preguntas como las siguientes:

¿Un mensaje de correo electrónico determinado es spam o no es spam?

¿Esta imagen es de un perro, o un gato?

Regresión lineal

Conceptos previos

La pendiente de una recta (m) indica que tan "empinada" esta la recta o ¿qué tanto incrementa o disminuye en Y con cada incremento en X?

m=incremento en y/incremento en

m=Δy/Δx

m=(y2-y1)/(x2-x1)

Una recta es un lugar geométrico de puntos tales que, para dos puntos cuales quiera (x1,y1) y (x2,y2) el valor de la pendiente (m) es constante.

m=(y2-y1)/(x2-x1)

Si se conoce un punto (x1,y1) de la recta y su pendiente, cualquier punto (x, y) se obtiene despejando la fórmula de la pendiente, tenemos:

(y-y1)=m(x-x1)

Suponga que (x1,y1) es (0,b) el punto donde la recta intersecta el eje Y, esto se llama ordenada al origen. Es decir, si sabemos que en la abscisa 0, la ordenada de la recta es b. Tenemos la formula:

(y-b)=m(x-0)

Despejando:

(y-b)=m(x)

Formula de la recta en forma: pendiente intersección:

y=mx+b

ecuación de la recta:

y=mx+b

Donde:
y es el valor para predecir
m es la pendiente
x es el valor de un atributo conocido
b es la intersección en y

En AA la ecuación dela recta
y=mx+b
se representa:
y=b+mx
sustituyendo m por w1 y b por w0

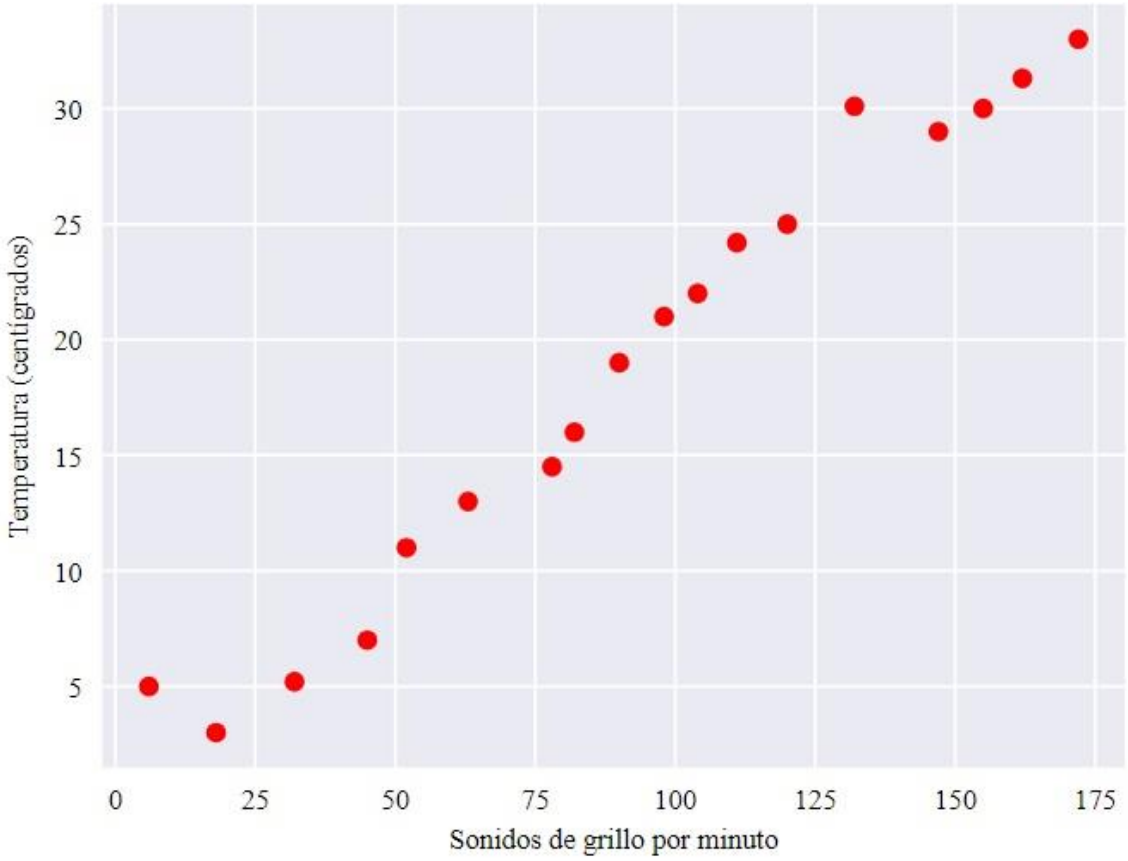
y'=w0+w1x1

Un modelo basado en 3 atributos sería:

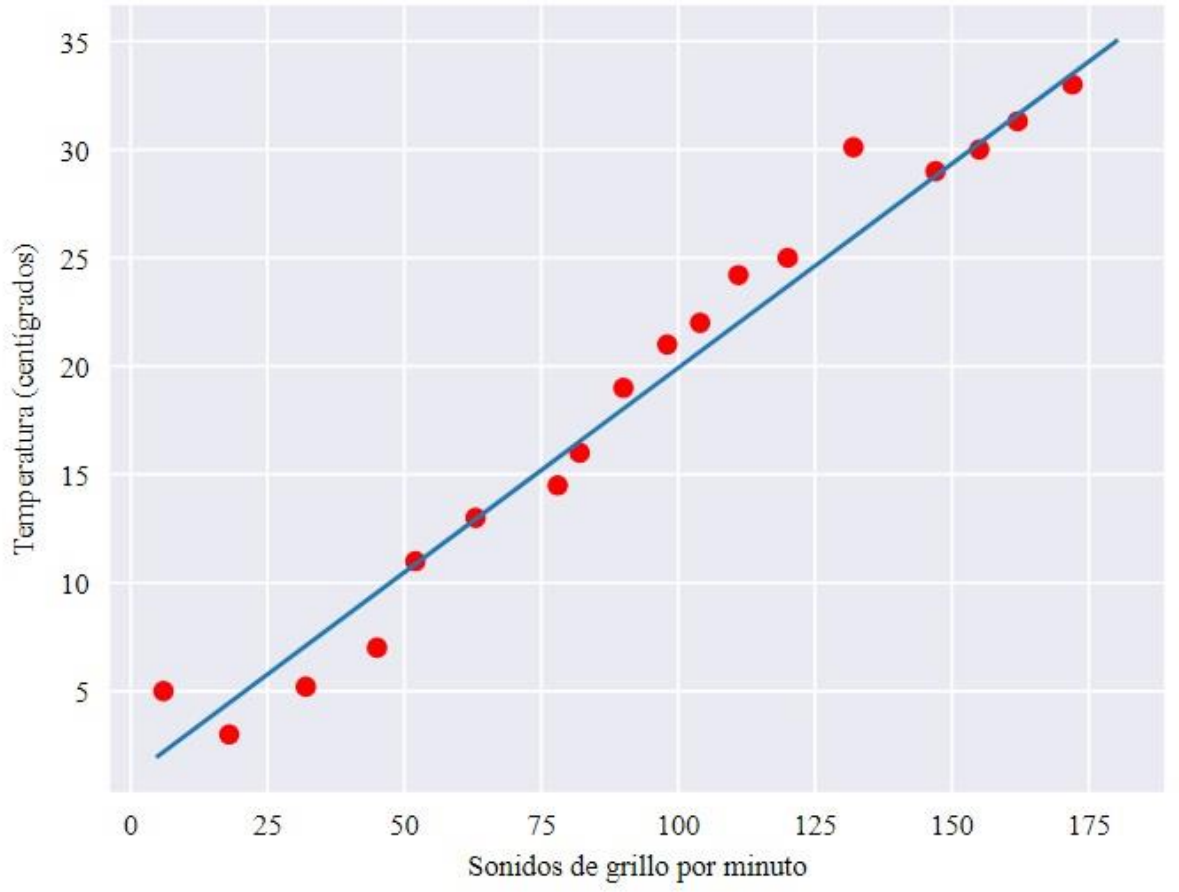
y'=w0+w1x1+w2x2+w3x3

Ejemplo de regresión lineal

Suponga que un entomólogo realizó las observaciones de cantos de grillo por minuto / temperatura del ambiente obteniendo los siguientes datos:



Y se observa una mayor cantidad de cantos a mayor temperatura, se puede observar que los datos tienen una relación lineal:



Si queremos predecir la temperatura con base en el número de cantos por minuto de los grillos, usando la fórmula de la recta tenemos:

$y'=mx+b$

Donde

y= es la temperatura que queremos predecir

m es la pendiente de la línea

x es la cantidad de cantos por minuto

b es la intersección de la recta en y.

Cambiando de notación:

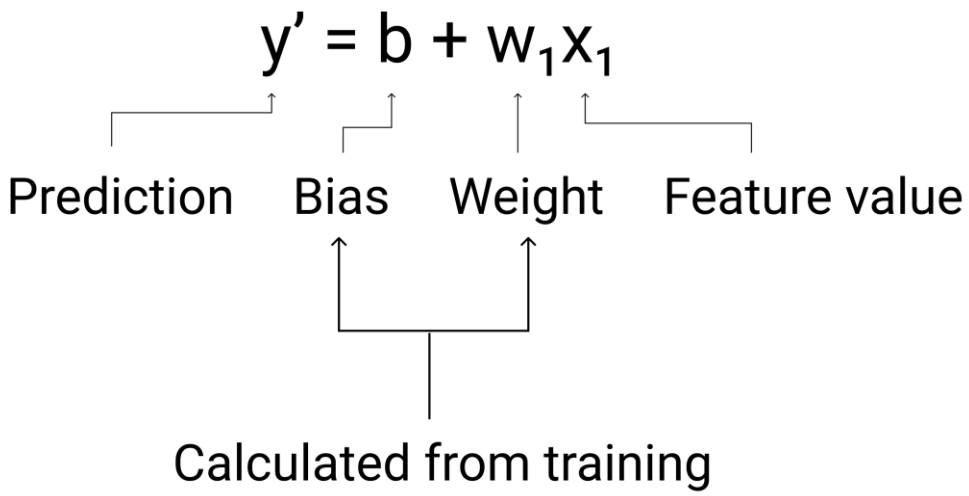
$y=w_0+w_1x_1$

donde $w_0=b$, $w_1=m$

Nota Usaremos esta notación en AA

A b (w0) se le suele llamar bias (inclinación o sesgo).
A w1 se le llama peso.
X característica.
Y predicción.

El peso y el sesgo son valores que actúan de parámetros en el aprendizaje.



Para modelos en más de dos dimensiones por ejemplo en 3D usaremos:

y=w0+w1x1+w2x2+w3x3

Sesgo. En aprendizaje automático (AA)

b (o w0) es el sesgo del modelo. El sesgo es el mismo concepto de la intersección en Y. En el AA, el sesgo suele denominarse w0. Sesgo es un parámetro del modelo y se calcula durante el entrenamiento.

w1 es el peso del punto. El peso es el mismo concepto de la pendiente m en la ecuación de una línea. El peso es un parámetro del modelo y es calcular durante el entrenamiento.

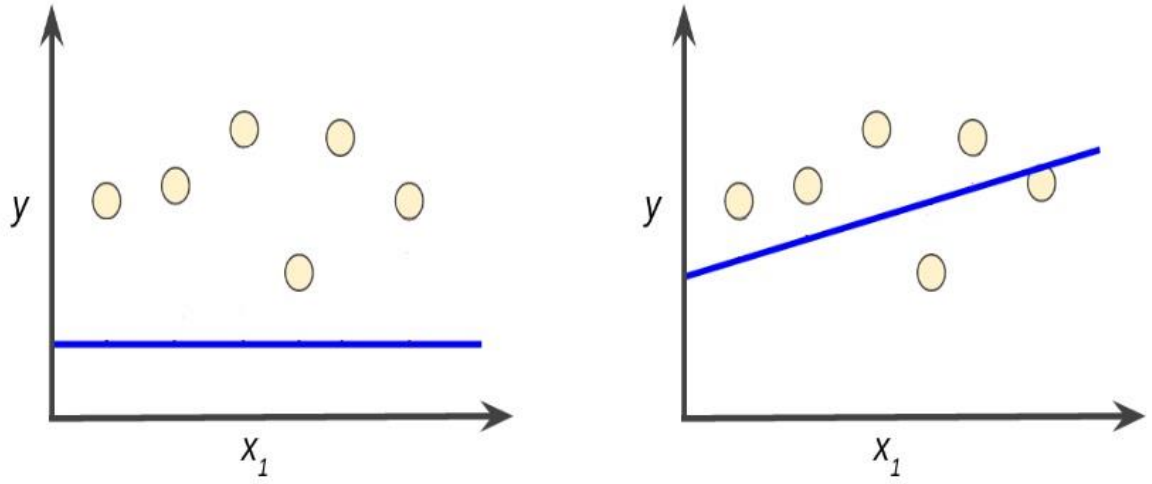
x1 es un atributo, de entrada.
Durante el entrenamiento, el modelo calcula el peso y el sesgo que producen un modelo.

Encontrar la recta

Para encontrar la recta puede usarse un proceso matemático (aunque a veces es muy complicado). Los métodos de AA sin embargo usan un enfoque iterativo, aquí usaremos el enfoque iterativo: trazar una recta aleatoria y ajustarla en cada iteración hasta obtener la mejor recta posible.

Para esto necesitamos un método que nos indique cuando una recta es "mejor" que otra, o se ajusta mejor a nuestro conjunto de datos.

¿En la siguiente imagen, cuál recta se ajusta mejor a nuestro conjunto de datos?:

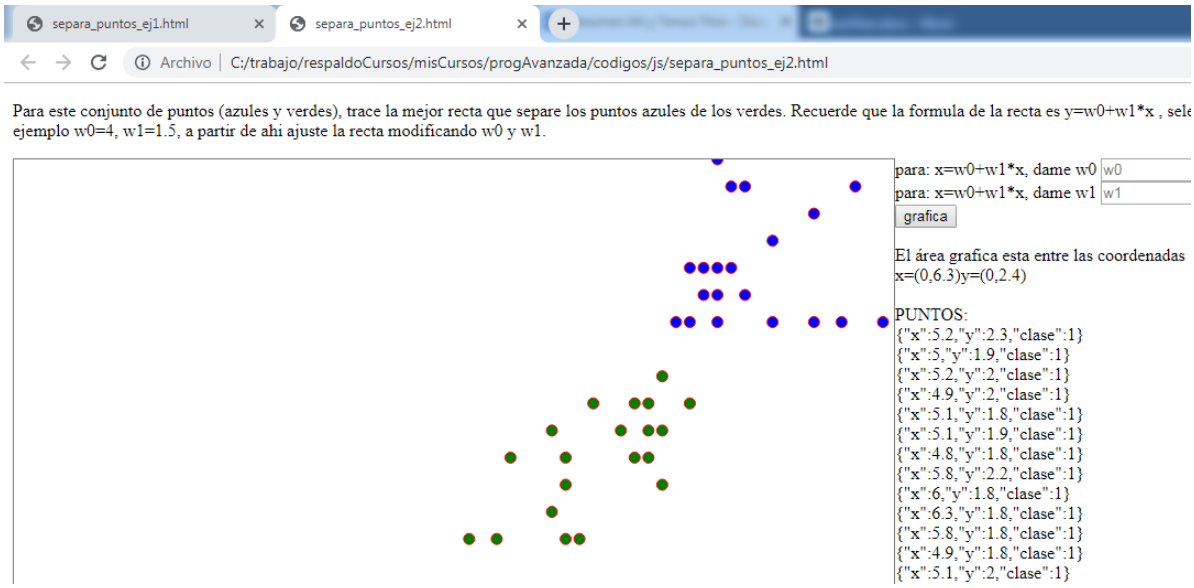


Claramente se ve que la segunda opción es la mejor, pero ¿cómo podemos calcularlo?

Ejercicio

Objetivo: entender como una recta puede ajustar un conjunto de puntos.

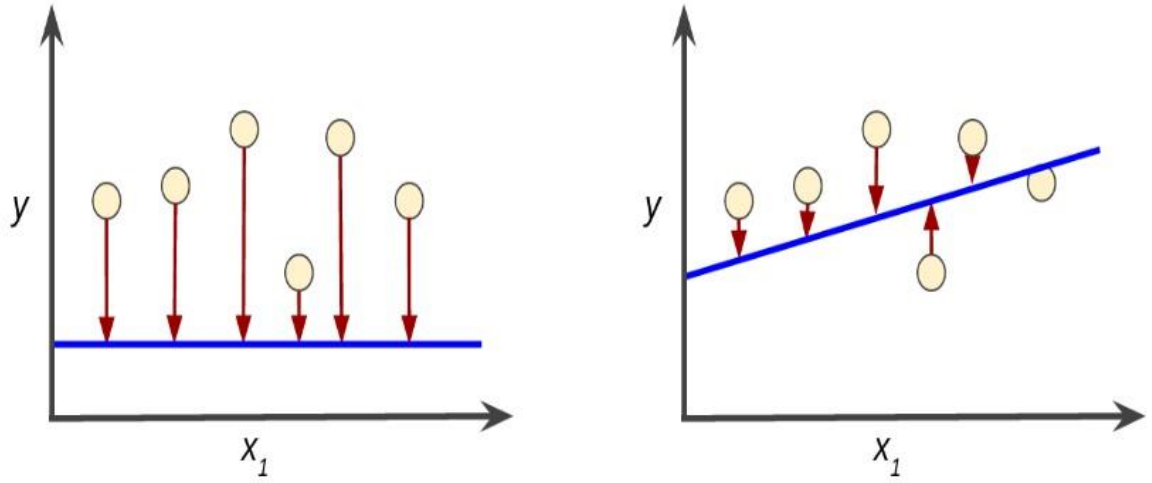
Actividades: en la liga: https://drive.google.com/open?id=1rtL1ob_2QbZV-fU51bu3m9met-my6bp9 encontrará un código en JS que muestra dos conjuntos de puntos como sigue:



Queremos encontrar una recta que separe ambos conjuntos de puntos (azules y verdes), para esto en los inputbox w0 y w1 debe poner un par de valores y probar si con esos valores obtiene la recta deseada. Empiece por ejemplo con w0=1 y w1=0.5, cambie los valores de w0 y w1 hasta obtener la recta adecuada.

Función de pérdida

En nuestro enfoque iterativo construimos varios modelos (varias rectas) y buscamos minimizar la pérdida. La pérdida es una penalización por una predicción incorrecta, es un número que indica que tan incorrecta es la predicción, necesitamos un modelo que minimice la pérdida, a esto se le llama minimización del riesgo empírico.



En esta imagen las líneas rojas representan la distancia entre la observación y (círculo beige) y la predicción (la línea azul). Se ve claramente que la imagen de la izquierda tiene una pérdida mayor que la imagen de la derecha.

Una forma de calcular la pérdida es realizar las sumas de la pérdida de cada observación contra su predicción. Sin embargo, como puede haber valores de pérdida positivos y negativos puede darse el caso en que la suma de cero pero haya mucha pérdida.

para evitar esto se utiliza la fórmula de error cuadrático medio ECM (***mean squared error***)

$$ECM = \frac{1}{N} \sum_{(x,y)\in D} (y - predicción(x))^2$$

donde:

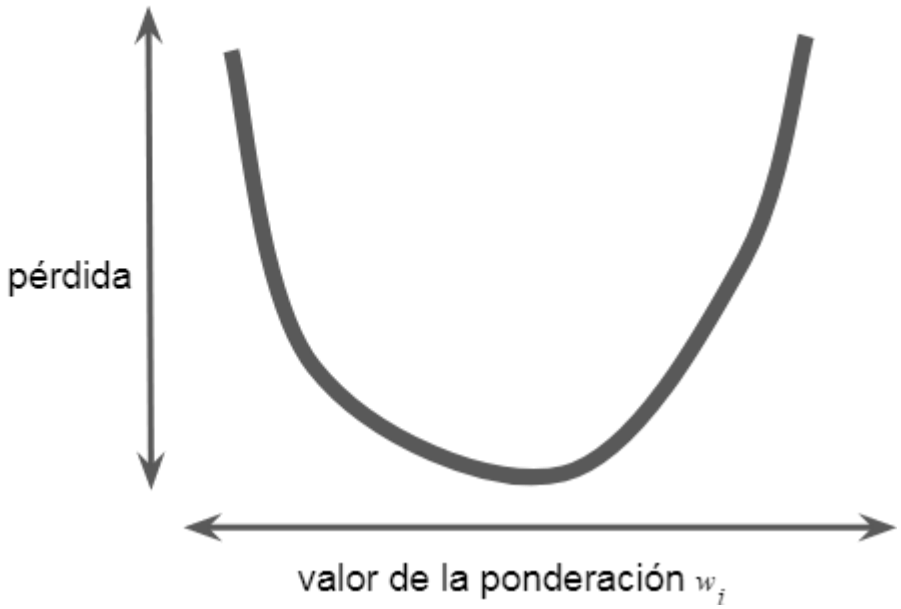
- (x,y) es un ejemplo en el que
  - x es el conjunto de atributos (p. ej., temperatura, edad y éxito para aparearse) que el modelo usa para realizar las predicciones.
  - y es la etiqueta del ejemplo (p. ej., cantos por minuto).
- predicción(x) es un atributo de las ponderaciones y las ordenadas al origen en combinación con el conjunto de atributos x.
- D es el conjunto de datos que contiene muchos ejemplos etiquetados, que son los pares .

- N es la cantidad de ejemplos en D.

Si bien **ECM** se usa comúnmente en el aprendizaje automático, no es la única función de pérdida práctica ni la mejor para todas las circunstancias.

Reducción de pérdida por descenso de gradiente

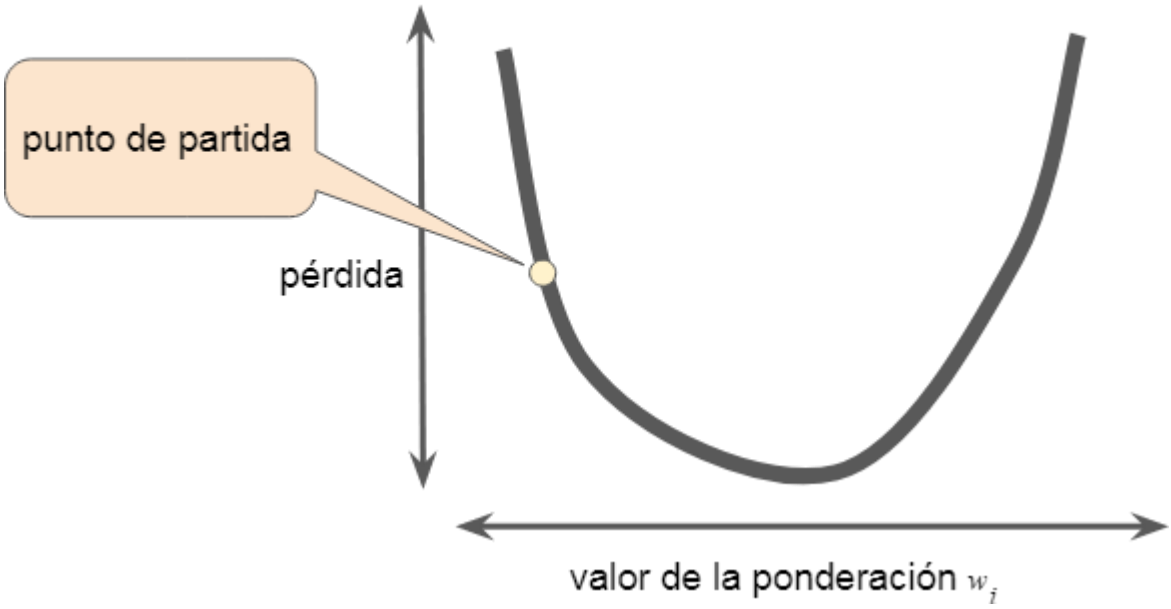
Supón que tuviéramos el tiempo y los recursos de cómputo para calcular la pérdida de todos los valores posibles de wi. Para el tipo de problemas de regresión que hemos estado examinando, la representación resultante de pérdida frente a wi siempre será convexa. En otras palabras, la representación siempre tendrá forma de tazón, como la siguiente:



Esta es una gráfica convexa. Los problemas convexos tienen un solo mínimo, es decir, un solo lugar en el que la pendiente es exactamente 0. Ese mínimo es donde converge la función de pérdida.

Calcular la función de pérdida para cada valor concebible de wi en todo el conjunto de datos sería una manera ineficaz de buscar el punto de convergencia. Examinemos un mecanismo más útil, muy popular en el aprendizaje automático, denominado descenso de gradientes.

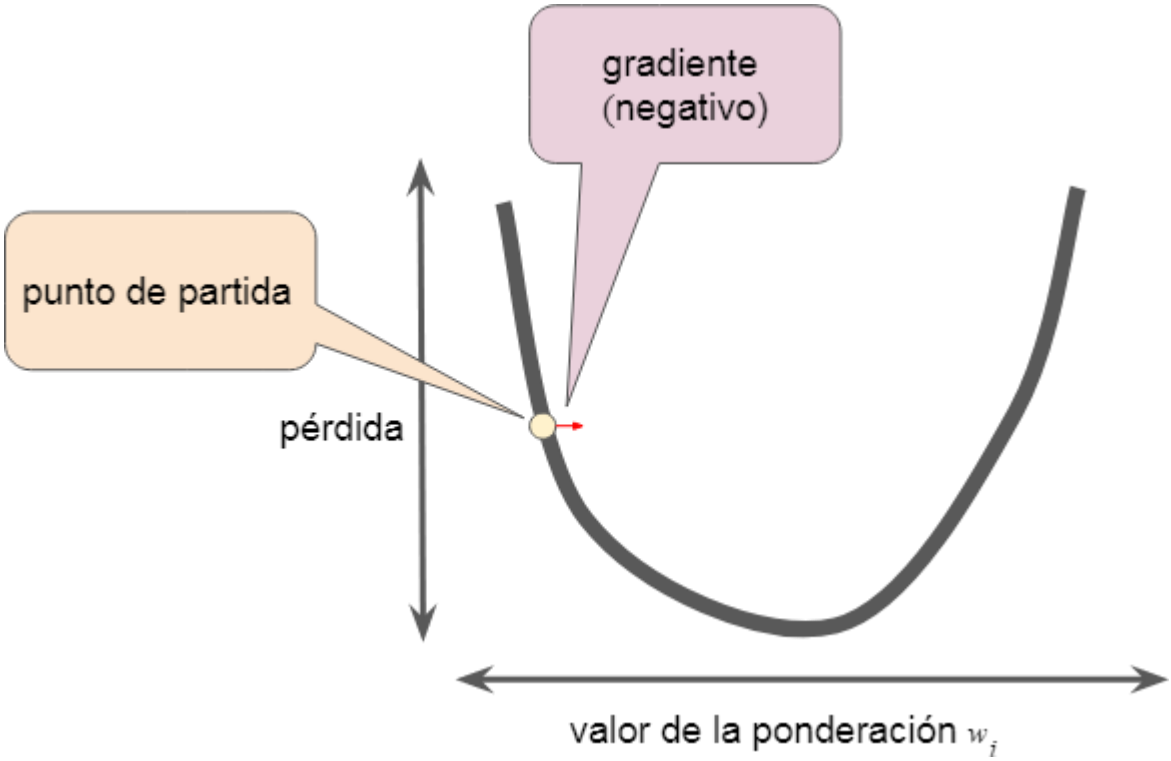
La primera etapa en el descenso de gradientes es elegir un valor de inicio (un punto de partida) para wi. El punto de partida no es muy importante; por lo tanto, muchos algoritmos simplemente establecen en 0 o eligen un valor al azar. En la siguiente figura, se muestra que elegimos un punto de partida levemente mayor que 0.



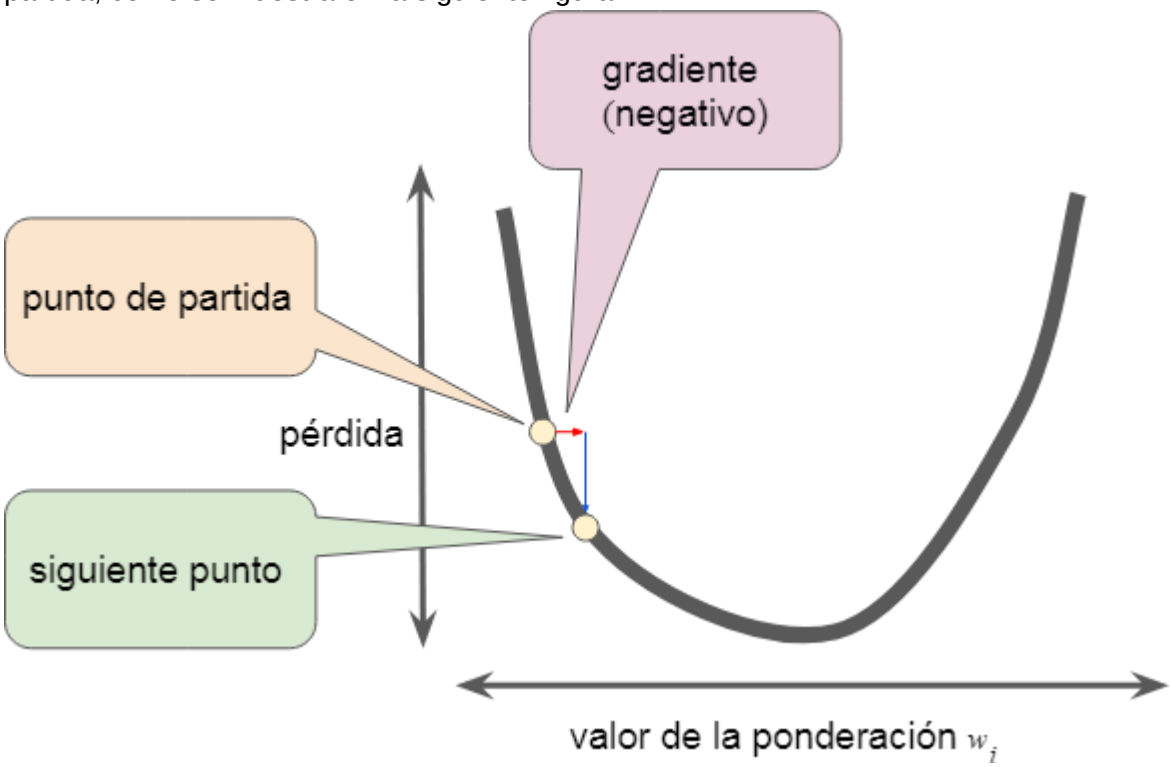
Luego, el algoritmo de descenso de gradientes calcula la gradiente de la curva de pérdida en el punto de partida. ***En resumen, un gradiente es un vector de derivadas parciales***; indica por dónde es más cerca o más lejos. Ten en cuenta que el gradiente de pérdida con respecto a un solo peso (como en la Figura 3) es equivalente a la derivada.
Ten en cuenta que el gradiente es un vector, de manera que tiene las dos características siguientes:

- una dirección
- una magnitud

El gradiente siempre apunta en la dirección del aumento más *empinado* de la función de pérdida. El algoritmo de descenso de gradientes toma un paso en dirección del gradiente negativo para reducir la pérdida lo más rápido posible.



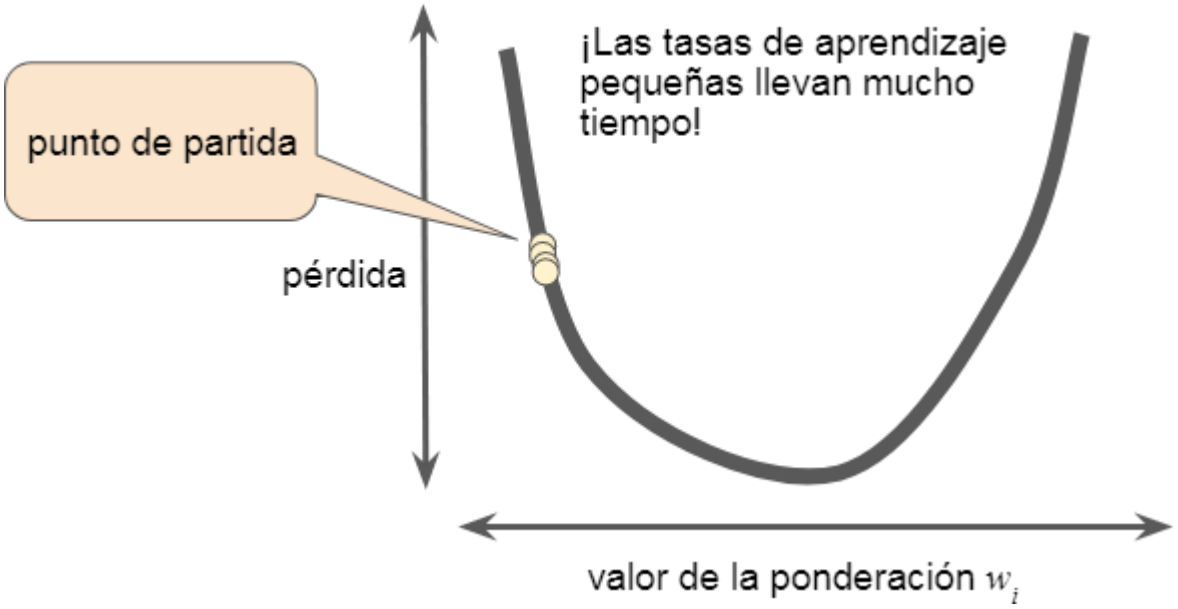
Para determinar el siguiente punto a lo largo de la curva de la función de pérdida, el algoritmo de descenso de gradientes agrega alguna fracción de la magnitud del gradiente al punto de partida, como se muestra en la siguiente figura:



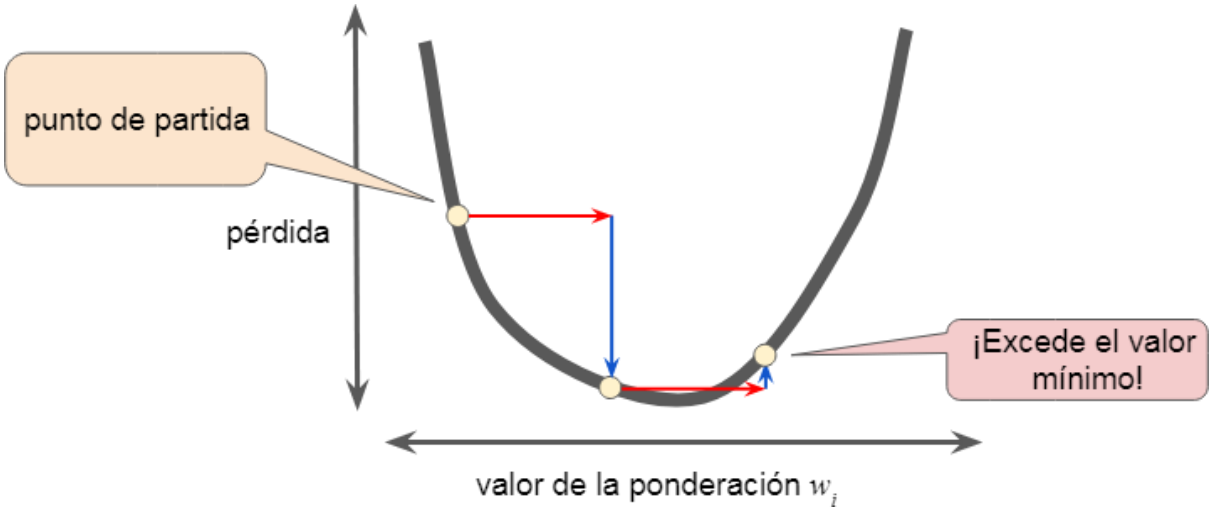
Tasa de aprendizaje

Como se observó, el vector de gradiente tiene una dirección y una magnitud. Los algoritmos de descenso de gradientes multiplican la gradiente por un escalar conocido como tasa de aprendizaje (o tamaño del paso en algunas ocasiones) para determinar el siguiente punto. Por ejemplo, si la magnitud de la gradiente es 2.5 y la tasa de aprendizaje es 0.01, el algoritmo de descenso de gradientes tomará el siguiente punto 0.025 más alejado del punto anterior.

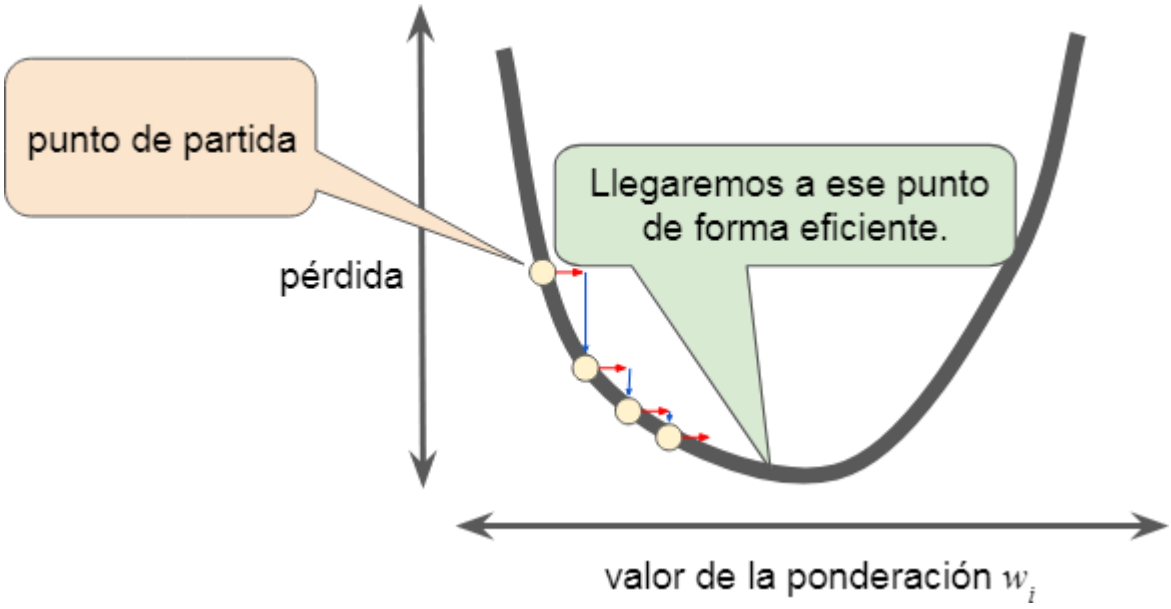
Los hiper parámetros son los controles que los programadores ajustan en los algoritmos de aprendizaje automático. La mayoría de los programadores de aprendizaje automático pasan gran parte de su tiempo ajustando la tasa de aprendizaje. Si eliges una tasa de aprendizaje muy pequeña, el aprendizaje llevará demasiado tiempo:



A la inversa, si especificas una tasa de aprendizaje muy grande, el siguiente punto rebotara al azar eternamente en la parte inferior, como un experimento de mecánica cuántica que salió muy mal:



Hay una tasa de aprendizaje con valor *dorado* para cada problema de regresión. El valor *dorado* está relacionado con qué tan plana es la función de pérdida. Si sabes que el gradiente de la función de pérdida es pequeño, usa una tasa de aprendizaje mayor, que compensará el gradiente pequeño y dará como resultado un tamaño del paso más grande.



Descenso de gradiente estocástico SGD

En el descenso de gradientes, ***un lote es la cantidad total de ejemplos que usas para calcular la gradiente en una sola iteración***. Hasta ahora, hemos supuesto que el lote era el conjunto de datos completo. Al trabajar a la escala de Google, los conjuntos de datos suelen tener miles de millones o incluso cientos de miles de millones de ejemplos. Además, los conjuntos de datos de Google con frecuencia contienen inmensas cantidades de atributos. En consecuencia, un lote puede ser enorme. Un lote muy grande puede causar que incluso una sola iteración tome un tiempo muy prolongado para calcularse.

Es probable que un conjunto de datos grande con ejemplos muestreados al azar contenga datos redundantes. De hecho, la redundancia se vuelve más probable a medida que aumenta el tamaño del lote. Un poco de redundancia puede ser útil para atenuar las gradientes inconsistentes, pero los lotes enormes tienden a no tener un valor mucho más predictivo que los lotes grandes.

¿Cómo sería si pudiéramos obtener la gradiente correcta en promedio con mucho menos cómputo? Al elegir ejemplos al azar de nuestro conjunto de datos, podríamos estimar (si bien de manera inconsistente) un promedio grande de otro mucho más pequeño. El descenso de gradiente estocástico (SGD) lleva esta idea al extremo: usa un solo ejemplo (un tamaño del lote de 1) por iteración. Cuando se dan demasiadas iteraciones, el SGD funciona, pero es muy inconsistente. El término "estocástico" indica que el ejemplo único que compone cada lote se elige al azar.

El descenso de gradiente estocástico de mini lote (SGD de mini lote) es un equilibrio entre la iteración de lote completo y el SGD. Un mini lote generalmente tiene entre 10 y 1,000 ejemplos, elegidos al azar. El SGD de mini lote reduce la cantidad de inconsistencia en el SGD, pero sigue siendo más eficaz que el lote completo.

Para simplificar la explicación, nos concentramos en el descenso de gradientes para un solo atributo. Te garantizamos que el descenso de gradientes también funciona en conjuntos de varios atributos.
Referencia:
https://developers.google.com/machine-learning/crash-course/reducing-loss/stochastic-gradient-descent?hl=es-419

Regresión logística

Muchos problemas requieren una estimación de probabilidad como salida. La regresión logística es un mecanismo extremadamente eficiente para calcular probabilidades. Hablando en términos prácticos, puede usar la probabilidad devuelta en cualquiera de las dos formas siguientes:

- "tal cual"
- Convertido a una categoría binaria.

Consideremos cómo podríamos usar la probabilidad "tal cual". Supongamos que creamos un modelo de regresión logística para predecir la probabilidad de que un perro ladre durante la mitad de la noche. Llamaremos a esa probabilidad:

p (ladrido | noche)

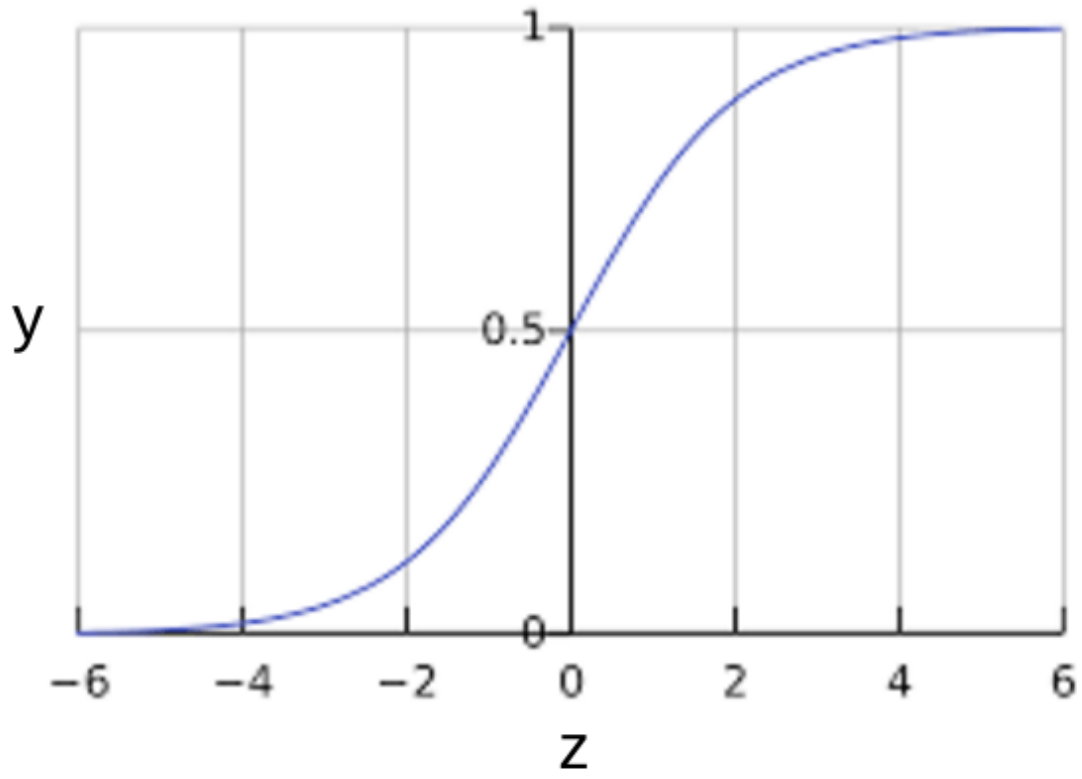
Si el modelo de regresión logística predice una ***p (ladrido | noche)*** de 0.05, entonces, durante un año, los dueños del perro deben sobresaltarse aproximadamente 18 veces:

*sobresalto= p (ladrido | noche) * noches*
*=0.05*365*
=18

En muchos casos, asignará la salida de regresión logística en la solución a un problema de clasificación binaria, en el que el objetivo es predecir correctamente una de las dos etiquetas posibles (por ejemplo, "spam" o "no spam"). Un módulo posterior se centra en eso.

Tal vez se pregunte cómo un modelo de regresión logística puede garantizar resultados que siempre se encuentren entre 0 y 1. Como sucede?, una función sigmoide, definida de la siguiente manera, produce resultados que tienen esas mismas características:

$y=1/(1+e^{-z})$



Si z representa la salida de la capa lineal de un modelo entrenado con regresión logística, entonces sigmoide (z) arrojará un valor (una probabilidad) entre 0 y 1. En términos matemáticos:

$y'=1/(1+e^{-z})$

dónde:

- y 'es la salida del modelo de regresión logística para un ejemplo particular.
- z=b+w1x1+w2x2+...+wNxN
    - Los valores de w son los pesos aprendidos del modelo, y b es el sesgo.
    - Los valores de x son los valores de característica para un ejemplo particular.

Función de pérdida para regresión logística

La función de pérdida para la regresión lineal es la pérdida al cuadrado. La función de pérdida para la regresión logística es la pérdida de registro, que se define de la siguiente manera:

$$\text{Log Loss} = \sum_{(x,y)\in D} -y\log(y') - (1-y)\log(1-y')$$

Donde:

- (x, y) en D es el conjunto de datos que contiene muchos ejemplos etiquetados, que son pares (x, y).
- y es la etiqueta en un ejemplo etiquetado. Como se trata de una regresión logística, cada valor de y debe ser 0 o 1.
- y' es el valor predicho (en algún lugar entre 0 y 1), dado el conjunto de características en x.

La ecuación para la pérdida de registro está estrechamente relacionada con la medida de entropía de Shannon de la teoría de la información. También es el logaritmo negativo de la función de verosimilitud, suponiendo una distribución de Bernoulli de y. De hecho, minimizar la función de pérdida produce una estimación de probabilidad máxima.

Regularización en regresión logística

La regularización es extremadamente importante en el modelado de regresión logística. Sin regularización, la naturaleza asintótica de la regresión logística seguiría impulsando la pérdida hacia 0 en altas dimensiones. En consecuencia, la mayoría de los modelos de regresión logística utilizan una de las dos estrategias siguientes para amortiguar la complejidad del modelo:

- L2 regularization.
- Pararse temprano, es decir, limitar el número de pasos de entrenamiento o la tasa de aprendizaje.

(Discutiremos una tercera estrategia, la regularización L1, en un módulo posterior).

Imagine que asigna una identificación única a cada ejemplo y asigna cada identificación a su propia función. Si no especifica una función de regularización, el modelo quedará completamente *sobre ajustado*. Esto se debe a que el modelo trataría de llevar la pérdida a cero en todos los ejemplos y nunca llegaría allí, llevando los pesos de cada función de indicador a + infinito o -infinito. Esto puede suceder en datos de alta dimensión con cruces de entidades, cuando hay una gran cantidad de cruces raros que ocurren solo en un ejemplo cada uno.

Afortunadamente, usar L2 o parar temprano evitará este problema.

Clasificación

La regresión logística devuelve una probabilidad. Puede usar la probabilidad devuelta "tal cual" (por ejemplo, la probabilidad de que el usuario haga clic en este anuncio es 0.00023) o convertir la probabilidad devuelta a un valor binario (por ejemplo, este correo electrónico es spam).

Clasificación: Verdadero vs.Falso y Positivo o Negativo

En esta sección, definiremos los bloques de construcción principales de las métricas que usaremos para evaluar los modelos de clasificación. Pero primero, una fábula:

Una fábula de Esopo: El niño que lloró por el lobo

Un pastor se aburre cuidando el rebaño del pueblo. Para divertirse, grita: "¡Lobo!" a pesar de que no hay ningún lobo a la vista. Los aldeanos corren para proteger al rebaño, pero luego se enojan mucho cuando se dan cuenta de que el niño les estaba jugando una broma.

Una noche, el pastorcillo ve a un lobo real acercarse al rebaño y grita: "¡Lobo!" Los aldeanos se niegan a ser engañados nuevamente y se quedan en sus casas. El lobo hambriento convierte el rebaño en chuletas de cordero. El pueblo pasa hambre. El pánico se produce.
Hagamos las siguientes definiciones:

- "Wolf" es una clase positiva.
- "No wolf" es una clase negativa.

Podemos resumir nuestro modelo de "predicción del lobo" utilizando una matriz de confusión 2x2 que representa los cuatro resultados posibles:

True Positive (TP): • Realidad: Un lobo amenazando. • Pastor dice: "Wolf." • Resultado: Pastor es un héroe.	False Positive (FP): • Realidad: No hay lobo amenazando. • Pastor dice: "Wolf." • Salida: Aldeanos enojados con el pastor por despertarlos.
False Negative (FN): • Realidad: Un lobo amenazando. • Pastor dice: "No wolf." • Salida: El lobo se come las ovejas.	True Negative (TN): • Realidad: No hay lobo amenazando. • Pastor dice: "No wolf." • Salida: Todo mundo bien.

Un verdadero positivo es un resultado en el que el modelo predice correctamente la clase positiva. Del mismo modo, un verdadero negativo es un resultado en el que el modelo predice correctamente la clase negativa.

Un falso positivo es un resultado en el que el modelo predice incorrectamente la clase positiva. Y un falso negativo es un resultado en el que el modelo predice incorrectamente la clase negativa.

En las siguientes secciones, veremos cómo evaluar los modelos de clasificación utilizando métricas derivadas de estos cuatro resultados.

Precisión

La precisión es una medida para evaluar los modelos de clasificación. Informalmente, la precisión es la fracción de predicciones que nuestro modelo acertó. Formalmente, la precisión tiene la siguiente definición:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Para la clasificación binaria, la precisión también se puede calcular en términos de positivos y negativos de la siguiente manera:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Donde TP = Positivos verdaderos, TN = Negativos verdaderos, FP = Positivos falsos y FN = Negativos falsos.

Intentemos calcular la precisión para el siguiente modelo que clasificó 100 tumores como malignos (la clase positiva) o benignos (la clase negativa):

True Positive (TP): • Reality: Malignant • ML model predicted: Malignant • Number of TP results: 1	False Positive (FP): • Reality: Benign • ML model predicted: Malignant • Number of FP results: 1
False Negative (FN): • Reality: Malignant • ML model predicted: Benign • Number of FN results: 8	True Negative (TN): • Reality: Benign • ML model predicted: Benign • Number of TN results: 90

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1 + 90}{1 + 90 + 1 + 8} = 0.91$$

La precisión llega a 0,91, o 91% (91 predicciones correctas de 100 ejemplos totales). Eso significa que nuestro clasificador de tumores está haciendo un gran trabajo al identificar tumores malignos, ¿verdad?

En realidad, hagamos un análisis más detallado de los aspectos positivos y negativos para obtener más información sobre el rendimiento de nuestro modelo.
De los 100 ejemplos de tumores, 91 son benignos (90 TN y 1 FP) y 9 son malignos (1 TP y 8 FN).

De los 91 tumores benignos, el modelo identifica correctamente 90 como benignos. Eso es bueno. Sin embargo, de los 9 tumores malignos, el modelo solo identifica correctamente 1 como maligno, un resultado terrible, ¡ya que 8 de 9 tumores malignos no se diagnostican!

Si bien la precisión del 91% puede parecer buena a primera vista, otro modelo clasificador de tumores que siempre predice benignos alcanzaría exactamente la misma precisión (91/100 predicciones correctas) en nuestros ejemplos. En otras palabras, nuestro modelo no es mejor que uno que tiene capacidad predictiva cero para distinguir tumores malignos de tumores benignos.

La precisión por sí sola no cuenta la historia completa cuando trabaja con un conjunto de datos con desequilibrio de clase, como este, donde hay una disparidad significativa entre el número de etiquetas positivas y negativas.

En la siguiente sección, veremos dos mejores métricas para evaluar problemas de desequilibrio de clase: precisión y recuperación.

Clasificación y recuerdo

Precisión intenta responder la siguiente pregunta:

¿Qué proporción de identificaciones positivas fue realmente correcta?

La precisión se define de la siguiente manera:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Calculemos la precisión para nuestro modelo ML de la sección anterior que analiza los tumores:

True Positives (TPs): 1	False Positives (FPs): 1
False Negatives (FNs): 8	True Negatives (TNs): 90

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{1}{1 + 1} = 0.5$$

Recall
Recall intenta responder la siguiente pregunta:

¿Qué proporción de positivos reales se identificó correctamente?

Matemáticamente, el recall se define de la siguiente manera:

Calculemos el recuerdo de nuestro clasificador tumoral:

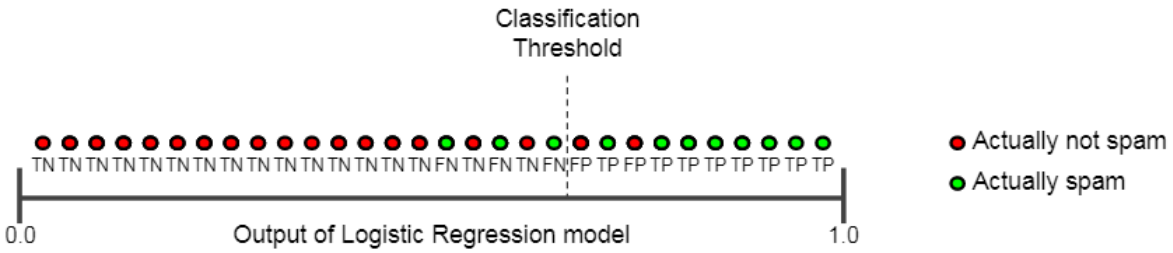
True Positives (TPs): 1	False Positives (FPs): 1
False Negatives (FNs): 8	True Negatives (TNs): 90

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{1}{1 + 8} = 0.11$$

Nuestro modelo tiene un recall de 0.11; en otras palabras, identifica correctamente el 11% de todos los tumores malignos.

Precisión y recall un tira y afloja

Para evaluar completamente la efectividad de un modelo, se debe examinar tanto la precisión como el recuerdo. Desafortunadamente, la precisión y el recuerdo a menudo están en tensión. Es decir, mejorar la precisión generalmente reduce el recuerdo y viceversa. Explore esta noción mirando la siguiente figura, que muestra 30 predicciones hechas por un modelo de clasificación de correo electrónico. Los que están a la derecha del umbral de clasificación se clasifican como "spam", mientras que los de la izquierda se clasifican como "no spam".



Calculemos la precisión y recall en función de los resultados que se muestran en la Figura anterior

True Positives (TP): 8	False Positives (FP): 2
False Negatives (FN): 3	True Negatives (TN): 17

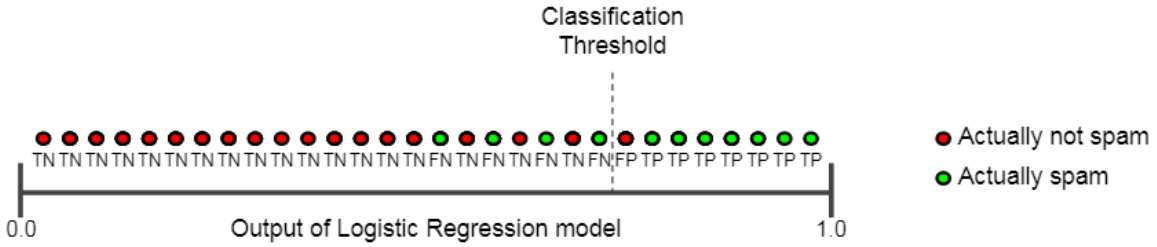
La precisión mide el porcentaje de correos electrónicos marcados como spam que se clasificaron correctamente, es decir, el porcentaje de puntos a la derecha de la línea de umbral que son verdes en la Figura

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{8}{8 + 2} = 0.8$$

La recuperación mide el porcentaje de correos electrónicos no deseados reales que se clasificaron correctamente, es decir, el porcentaje de puntos verdes que están a la derecha de la línea de umbral en la Figura

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{8}{8 + 3} = 0.73$$

Cambiemos el umbral de clasificación:



El número de falsos positivos disminuye, pero los falsos negativos aumentan. Como resultado, la precisión aumenta, mientras que la recuperación disminuye

True Positives (TP): 7	False Positives (FP): 1
False Negatives (FN): 4	True Negatives (TN): 18

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{7}{7 + 1} = 0.88$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{7}{7 + 4} = 0.64$$

Se han desarrollado varias métricas que dependen tanto de la precisión como del recall.

Umbral (Thresholding)

Un modelo de regresión logística que devuelve 0.9995 para un mensaje de correo electrónico en particular predice que es muy probable que sea spam. Por el contrario, es muy probable que otro mensaje de correo electrónico con un puntaje de predicción de 0,0003 en el mismo modelo de regresión logística no sea spam. Sin embargo, ¿qué pasa con un mensaje de correo electrónico con un puntaje de predicción de 0.6? Para asignar un valor de regresión logística a una categoría binaria, se debe definir un umbral de clasificación (también llamado umbral de decisión). Un valor por encima de ese umbral indica "spam"; un valor a continuación indica "no es spam". Es tentador suponer que el umbral de clasificación siempre debe ser 0.5, pero los umbrales dependen del problema y, por lo tanto, son valores que debe ajustar.

Clasificación: sesgo de predicción

Las predicciones de regresión logística deben ser imparciales. Es decir:

"promedio de predicciones" debe ≈ "promedio de observaciones"

El sesgo de predicción es una cantidad que mide qué tan separados están esos dos promedios. Es decir:

prediction bias = promedio de predicciones - promedio de observaciones.

Nota: El "sesgo de predicción" es una cantidad diferente al sesgo (la b en wx + b).

Un sesgo de predicción significativo distinto de cero le indica que hay un error en algún lugar de su modelo, ya que indica que el modelo está equivocado acerca de la frecuencia con la que se producen las etiquetas positivas. Por ejemplo, supongamos que sabemos que, en promedio, el 1% de todos los correos electrónicos son spam. Si no sabemos nada acerca de un correo electrónico determinado, debemos predecir que es probable que sea 1% spam. Del mismo modo, un buen modelo de spam debería predecir en promedio que los correos electrónicos tienen un 1% de probabilidades de ser spam. (En otras palabras, si promediamos las probabilidades pronosticadas de que cada correo electrónico individual sea spam, el resultado debería ser del 1%). Si, en cambio, la predicción promedio del modelo es del 20% de probabilidad de ser spam, podemos concluir que muestra un sesgo de predicción.

Las posibles causas raíz del sesgo de predicción son:

- Conjunto de características incompletas
- Conjunto de datos ruidosos
- Tubería de Buggy

- Muestra de entrenamiento sesgada
- Regularización demasiado fuerte

Es posible que sienta la tentación de corregir el sesgo de predicción procesando posteriormente el modelo aprendido, es decir, agregando una capa de calibración que ajuste la salida de su modelo para reducir el sesgo de predicción. Por ejemplo, si su modelo tiene un sesgo de + 3%, podría agregar una capa de calibración que reduzca la predicción media en un 3%. Sin embargo, agregar una capa de calibración es una mala idea por las siguientes razones:

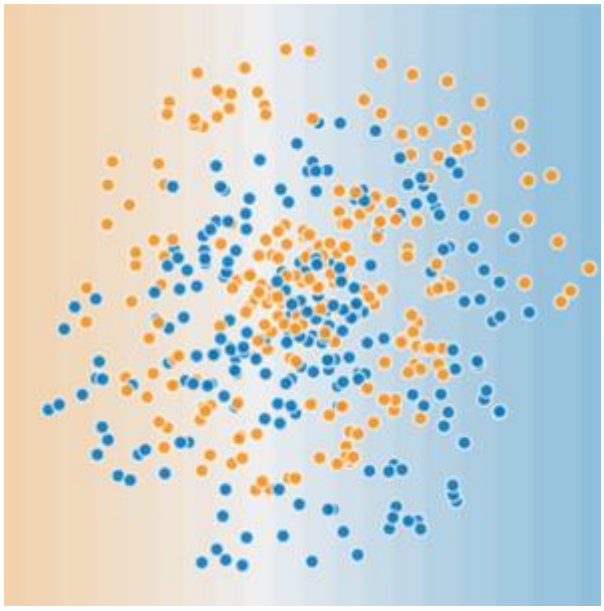
- Estás arreglando el síntoma en lugar de la causa.
- Has creado un sistema más frágil que ahora debes mantener actualizado.

Nota: Un buen modelo generalmente tendrá un sesgo cercano a cero. Dicho esto, un sesgo de predicción bajo no prueba que su modelo sea bueno. Un modelo realmente terrible podría tener un sesgo de predicción cero. Por ejemplo, un modelo que solo predice el valor medio para todos los ejemplos sería un mal modelo, a pesar de tener un sesgo cero.

La regresión logística predice un valor entre 0 y 1. Sin embargo, todos los ejemplos etiquetados son exactamente 0 (que significa, por ejemplo, "no spam") o exactamente 1 (que significa, por ejemplo, "spam"). Por lo tanto, al examinar el sesgo de predicción, no puede determinar con precisión el sesgo de predicción basándose en un solo ejemplo; debe examinar el sesgo de predicción en un "cubo" de ejemplos. Es decir, el sesgo de predicción para la regresión logística sólo tiene sentido cuando se agrupan suficientes ejemplos para poder comparar un valor pronosticado (por ejemplo, 0.392) con los valores observados (por ejemplo, 0.394).

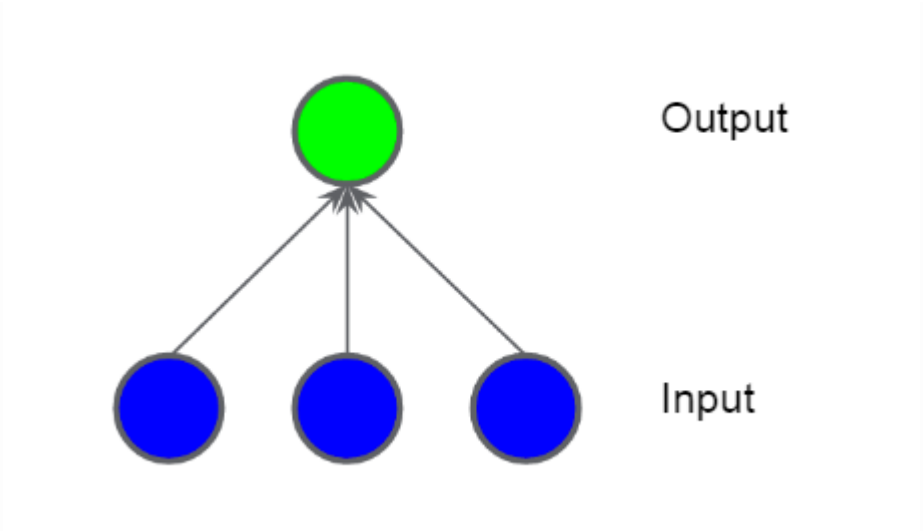
Red neuronal

Para problemas no lineales como:



Puntos no linealmente separables

Las redes neuronales pueden ayudar con problemas no lineales, comencemos por representar un modelo lineal como un gráfico:

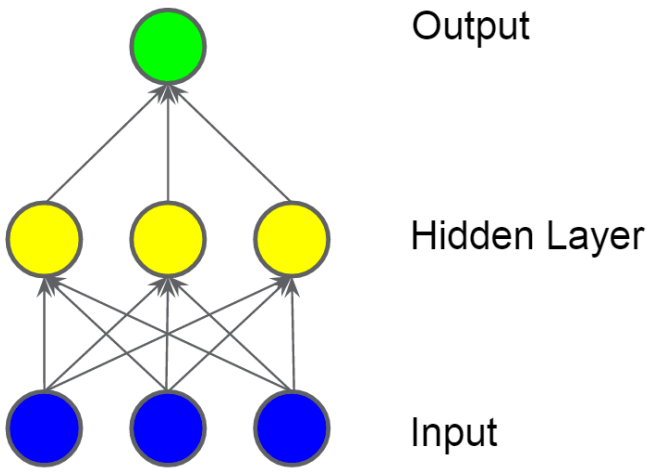


Cada círculo azul representa una característica de entrada, y el círculo verde representa la suma ponderada de las entradas.

¿Cómo podemos alterar este modelo para mejorar su capacidad para lidiar con problemas no lineales?

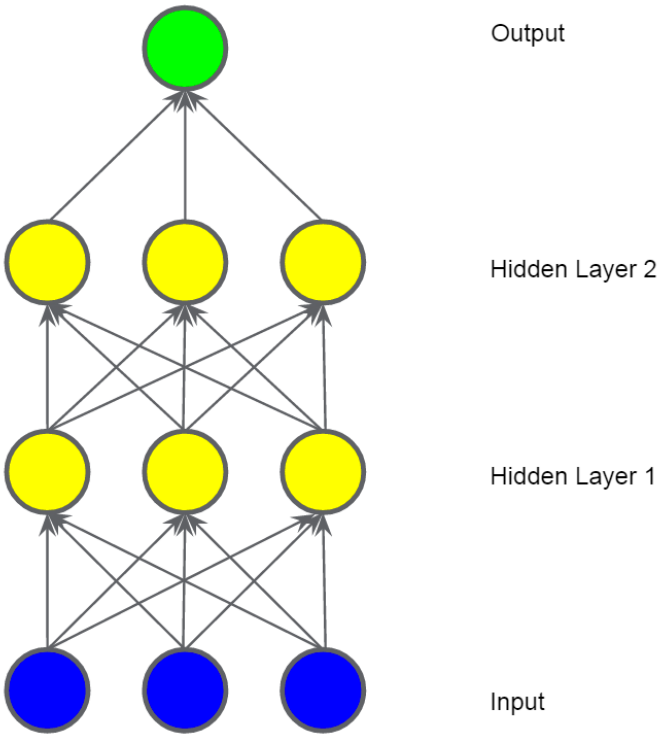
Capas ocultas

En el modelo representado por el siguiente gráfico, hemos agregado una "capa oculta" de valores intermedios. Cada nodo amarillo en la capa oculta es una suma ponderada de los valores del nodo de entrada azul. La salida es una suma ponderada de los nodos amarillos.



¿Es este modelo lineal? Sí, su salida sigue siendo una combinación lineal de sus entradas.

En el modelo representado por el siguiente gráfico, hemos agregado una segunda capa oculta de sumas ponderadas.

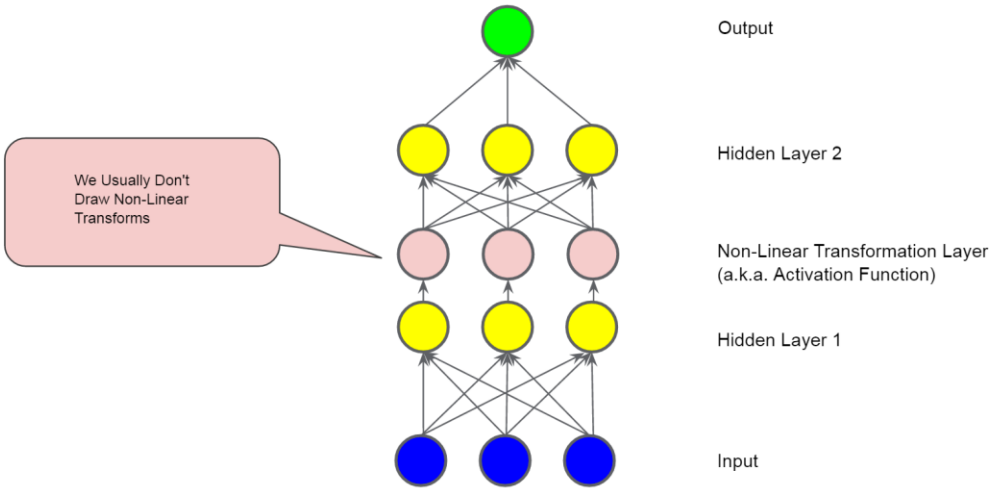


¿Este modelo sigue siendo lineal? Sí lo es. Cuando expresa la salida como una función de la entrada y simplifica, obtiene solo otra suma ponderada de las entradas. Esta suma no modelará efectivamente el problema no lineal en la Figura que muestra los puntos no linealmente separables.

Función de activación.

Para modelar un problema no lineal, podemos introducir directamente una no linealidad. Podemos canalizar cada nodo de capa oculta a través de una función no lineal.

En el modelo representado por el siguiente gráfico, el valor de cada nodo en la capa oculta 1 se transforma mediante una función no lineal antes de pasar a las sumas ponderadas de la siguiente capa. Esta función no lineal se llama función de activación.

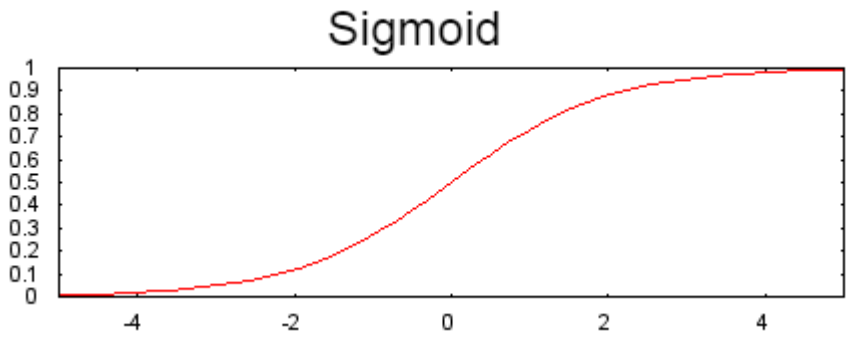


Ahora que hemos agregado una función de activación, agregar capas tiene más impacto. Apilar las no linealidades en las no linealidades nos permite modelar relaciones muy complicadas entre las entradas y las salidas predichas. En resumen, cada capa está aprendiendo efectivamente una función más compleja y de alto nivel sobre las entradas sin procesar. Si desea desarrollar más intuición sobre cómo funciona esto, vea la excelente publicación de blog de Chris Olah: https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/ .

Funciones comunes de activación.

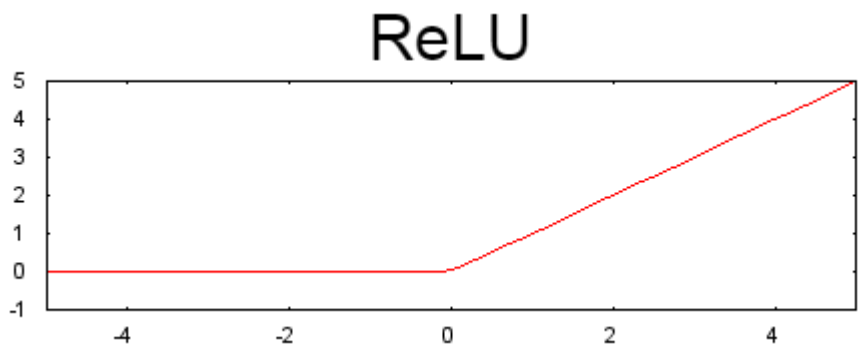
La siguiente función de activación sigmoidea convierte la suma ponderada en un valor entre 0 y 1.

$F(x)=1/(1-e^{-x})$



La siguiente función de activación de unidad lineal rectificada (o ReLU, para abreviar) a menudo funciona un poco mejor que una función suave como el sigmoide, a la vez que es significativamente más fácil de calcular.

$F(x)=max(0,x)$



De hecho, cualquier función matemática puede servir como una función de activación. Supongamos que representa nuestra función de activación (Relu, Sigmoid o lo que sea). En consecuencia, el valor de un nodo en la red viene dado por la siguiente fórmula:
$\partial(wx+b)$

TensorFlow proporciona compatibilidad inmediata para una amplia variedad de funciones de activación. Dicho esto, todavía recomendamos comenzar con ReLU.

Mejores prácticas

Esta sección explica los casos de falla de propagación hacia atrás y la forma más común de regularizar una red neuronal.

Hay varias formas comunes de que la retropropagación salga mal. Los gradientes para las capas inferiores (más cercanas a la entrada) pueden volverse muy pequeños. En redes profundas, calcular estos gradientes puede implicar tomar el producto de muchos términos pequeños. Cuando los gradientes tienden a 0 para las capas inferiores, estas capas se entrenan muy lentamente, o nada en absoluto.

La función de activación ReLU puede ayudar a prevenir la desaparición de gradientes.

Si los pesos en una red son muy grandes, entonces los gradientes para las capas inferiores involucran productos de muchos términos grandes. En este caso, puede tener gradientes explosivos: gradientes que se vuelven demasiado grandes para converger.

La normalización por lotes puede ayudar a evitar la explosión de gradientes, al igual que puede reducir la tasa de aprendizaje.

Una vez que la suma ponderada de una unidad ReLU cae por debajo de 0, la unidad ReLU puede atascarse. Produce 0 activaciones, sin aportar nada a la salida de la red, y los gradientes ya no pueden fluir a través de ella durante la propagación hacia atrás. Con una fuente de gradientes cortada, la entrada a ReLU puede no cambiar lo suficiente como para que la suma ponderada vuelva a estar por encima de 0.

Bajar la tasa de aprendizaje puede ayudar a evitar que las unidades ReLU mueran.
Otra forma de regularización, llamada Dropout, es útil para las redes neuronales. Funciona cuando al azar "dropping out" (abandonamos) aleatoriamente las activaciones de la unidad

en una red para un solo paso de gradiente. Cuanto más abandones, más fuerte será la regularización:

- 0.0 = Sin regularización de abandono.
- 1.0 = Eliminar todo. El modelo no aprende nada.
- Valores entre 0.0 y 1.0 = Más útil.

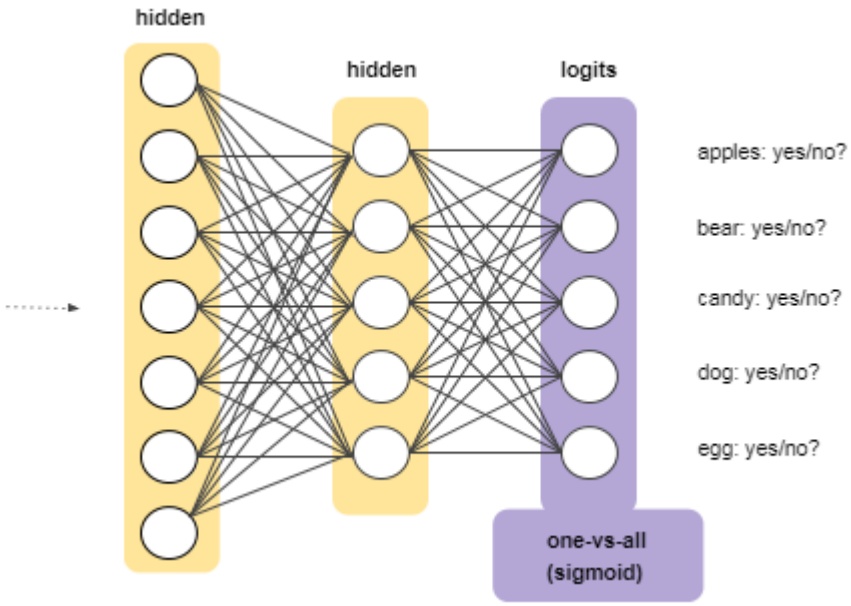
Redes multicapas One vs. all

proporciona una manera de aprovechar la clasificación binaria. Dado un problema de clasificación con N posibles soluciones, una solución de uno contra todos consiste en N clasificadores binarios separados, un clasificador binario para cada resultado posible. Durante el entrenamiento, el modelo recorre una secuencia de clasificadores binarios, entrenando a cada uno para responder una pregunta de clasificación por separado. Por ejemplo, dada una imagen de un perro, se pueden entrenar cinco reconocedores diferentes, cuatro ven la imagen como un ejemplo negativo (no un perro) y uno ve la imagen como un ejemplo positivo (un perro). Es decir:

- ¿Es esta imagen una manzana? No.
- ¿Es esta imagen un oso? No.
- ¿Es esta imagen dulce? No.
- ¿Es esta imagen un perro? Si.
- ¿Es esta imagen un huevo? No.

Este enfoque es bastante razonable cuando el número total de clases es pequeño, pero se vuelve cada vez más ineficiente a medida que aumenta el número de clases.

Podemos crear un modelo uno contra todo significativamente más eficiente con una red neuronal profunda en la que cada nodo de salida representa una clase diferente. La siguiente figura sugiere este enfoque:



Softmax

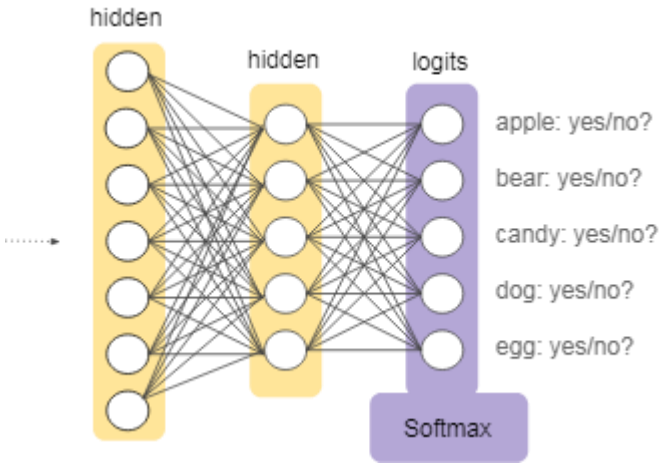
Recuerde que la regresión logística produce un decimal entre 0 y 1.0. Por ejemplo, una salida de regresión logística de 0.8 de un clasificador de correo electrónico sugiere un 80% de probabilidad de que un correo electrónico sea spam y un 20% de probabilidad de que no sea spam. Claramente, la suma de las probabilidades de que un correo electrónico sea spam o no spam es 1.0.

Softmax extiende esta idea a un mundo de múltiples clases. Es decir, Softmax asigna probabilidades decimales a cada clase en un problema de varias clases. Esas probabilidades decimales deben sumar 1.0. Esta restricción adicional ayuda a que el entrenamiento converja más rápidamente de lo que lo haría de otra manera.

Por ejemplo, volviendo al análisis de imagen que vimos en la Figura anterior, Softmax podría producir las siguientes probabilidades de que una imagen pertenezca a una clase particular:

Class	Probability
apple	0.001
bear	0.04
candy	0.008
dog	0.95
egg	0.001

Softmax se implementa a través de una capa de red neuronal justo antes de la capa de salida. La capa Softmax debe tener el mismo número de nodos que la capa de salida.



Opciones de softmax

Considere las siguientes variantes de Softmax:

- Softmax completo es el Softmax que hemos estado discutiendo; es decir, Softmax calcula una probabilidad para cada clase posible.
- El muestreo de candidatos significa que Softmax calcula una probabilidad para todas las etiquetas positivas pero solo para una muestra aleatoria de etiquetas negativas. Por ejemplo, si estamos interesados en determinar si una imagen de entrada es un beagle o un sabueso, no tenemos que proporcionar probabilidades para cada ejemplo que no sea perrito.

Full Softmax es bastante barato cuando el número de clases es pequeño, pero se vuelve prohibitivamente caro cuando el número de clases aumenta. El muestreo de candidatos puede mejorar la eficiencia en problemas que tienen un gran número de clases.

Softmax asume que cada ejemplo es miembro de exactamente una clase. Sin embargo, algunos ejemplos pueden ser simultáneamente miembros de múltiples clases. Para tales ejemplos:

- No puede usar Softmax.
- Debe confiar en múltiples regresiones logísticas.

Por ejemplo, suponga que sus ejemplos son imágenes que contienen exactamente un elemento: una pieza de fruta. Softmax puede determinar la probabilidad de que ese elemento sea una pera, una naranja, una manzana, etc. Si sus ejemplos son imágenes que contienen todo tipo de cosas (cuencos de diferentes tipos de fruta), tendrá que usar múltiples regresiones logísticas.

aqui me quede:
https://developers.google.com/machine-learning/crash-course/embeddings/motivation-from-collaborative-filtering

Tensor Flow

Tensoflow es un framework para realizar cálculos usando tensores en JS. Un tensor es la generalización de un vector o matriz en altas dimensiones. La unidad central de datos en TF son los tf.Tensor que es un conjunto de valores conformado en un arreglo de 1 o más dimensiones.

Regresión lineal con TF

Veamos un ejemplo de su uso obtenido de:
https://codelabs.developers.google.com/codelabs/tfjs-training-regression/index.html#0

En este ejemplo entrenaremos un modelo para hacer predicciones sobre datos que describen un conjunto de autos, este es un problema de regresión lineal.

Google tiene disponible un conjunto de datos de prueba que contiene iinformación sobre autos, esta en formato JSON y la puede obtener desde la liga:
https://storage.googleapis.com/tfjs-tutorials/carsData.json

Escriba esta liga en su navegador, deberá ver en su ventana un conjunto de datos json como el siguiente:


```
[
  {
    "Name": "chevrolet chevelle malibu",
    "Miles_per_Gallon": 18,
    "Cylinders": 8,
    "Displacement": 307,
    "Horsepower": 130,
    "Weight_in_lbs": 3504,
    "Acceleration": 12,
    "Year": "1970-01-01",
    "Origin": "USA"
  },
  {
    "Name": "buick skylark 320",
    "Miles_per_Gallon": 15,
    "Cylinders": 8,
    "Displacement": 350,
    "Horsepower": 165,
    "Weight_in_lbs": 3693,
    "Acceleration": 11.5,
    "Year": "1970-01-01",
    "Origin": "USA"
  },
…
]
```


Leer los datos.

Ahora escribimos un código que lea estos datos desde js, cómo usaremos la función fetch se requiere que este ejemplo corra sobre un servidor, para este ejemplo tengo instalado WAMP SERVER, requerimos los siguientes 2 archivos que yo guarde en la carpeta www/pruebas/pruebas-tf:

archivo:index.html

```
<!DOCTYPE html>
<html>
<head>
```

```
  <title>tutorial TensorFlow.js </title>

  <!-- Import TensorFlow.js -->
  <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@1.0.0/dist/tf.min.js"></script>
  <!-- Import tfjs-vis -->
  <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis@1.0.2/dist/tfjs-vis.umd.min.js"></script>

  <script src="script.js"></script>

</head>

<body>
</body>
</html>
```

archivo: srcript.js

```
async function getData() {
    const carsDataReq = await fetch('https://storage.googleapis.com/tfjs-tutorials/carsData.json');
    const carsData = await carsDataReq.json();
    const cleaned = carsData.map(car => ({
      mpg: car.Miles_per_Gallon,
      horsepower: car.Horsepower,
    }))
    .filter(car => (car.mpg != null && car.horsepower != null));
    document.write (JSON.stringify(cleaned));
    return cleaned;
  }

document.addEventListener('DOMContentLoaded', getData);
```

Al probar el ejemplo desde su navegador: http://localhost/pruebas/pruebas-tf/index.html verá datos como los siguientes:

```
[{"mpg":18,"horsepower":130},{"mpg":15,"horsepower":165},...
```

Cómo ve la función getData filtro solo 2 atributos del archivo de entrada: millas por galón (mpg) y caballos de fuerza.

Nota para entender mejor este código vea los métodos map en:
https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/map
y filter en:
https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/filter

Graficando los datos.

Cambiaremos el contenido del archivo script.js por el siguiente:

archivo: script.js

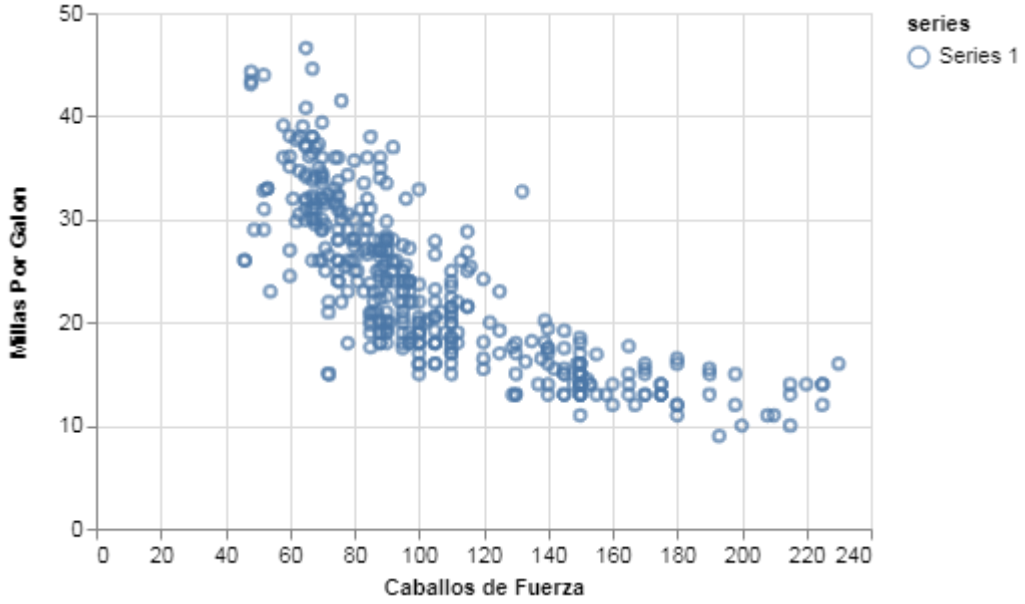
```
async function getData() {
   const carsDataReq = await fetch('https://storage.googleapis.com/tfjs-tutorials/carsData.json');
   const carsData = await carsDataReq.json();
   const cleaned = carsData.map(car => ({
     mpg: car.Miles_per_Gallon,
     horsepower: car.Horsepower,
   }))
   .filter(car => (car.mpg != null && car.horsepower != null));
   return cleaned;
 }
 async function run() {
   // Load and plot the original input data that we are going to train on.
   const data = await getData();
   const values = data.map(d => ({
     x: d.horsepower,
     y: d.mpg,
   }));

   tfvis.render.scatterplot(
     {name: 'caballos de fuerza vs Millas Por Galon'},
     {values},
     {
       xLabel: 'Caballos de Fuerza',
       yLabel: 'Millas Por Galon',
       height: 300
     }
   );

   // More code will be added below
 }

 document.addEventListener('DOMContentLoaded', run);
```

Verá la siguiente gráfica:



Modelo

En esta sección escribiremos código para describir la arquitectura del modelo. La arquitectura del modelo es solo una forma elegante de decir "qué funciones ejecutará el modelo cuando se esté ejecutando", o alternativamente "qué algoritmo utilizará nuestro modelo para calcular sus respuestas".
Los modelos Machine Learning (ML) son algoritmos que toman una entrada y producen una salida. Cuando se utilizan redes neuronales, el algoritmo es un conjunto de capas de neuronas con "pesos" (números) que rigen su salida. El proceso de capacitación aprende los valores ideales para esos pesos.

Agregue el siguiente código a su archivo script.js:

```
function createModel() {
  // Create a sequential model
  const model = tf.sequential();

  // Add a single hidden layer
  model.add(tf.layers.dense({inputShape: [1], units: 1, useBias: true}));

  // Add an output layer
  model.add(tf.layers.dense({units: 1, useBias: true}));

  return model;
}
```

Este es uno de los modelos más simples que podemos definir en tensorflow.js, analicemos un poco cada línea.

Instanciar el modelo:
const model = tf.sequential();

Esto crea una instancia de un objeto tf.Model. Este modelo es secuencial porque sus entradas fluyen directamente hacia su salida. Otros tipos de modelos pueden tener ramificaciones, o incluso múltiples entradas y salidas, pero en muchos casos sus modelos serán secuenciales. Los modelos secuenciales también tienen una API más fácil de usar.

Agregar capas
model.add(tf.layers.dense({inputShape: [1], units: 1, useBias: true}));

Esto agrega una capa oculta a nuestra red. Una capa densa es un tipo de capa que multiplica sus entradas por una matriz (llamada pesos) y luego agrega un número (llamado sesgo) al resultado. Como esta es la primera capa de la red, necesitamos definir nuestra inputShape. InputShape es [1] porque tenemos 1 número como entrada (los caballos de fuerza de un automóvil dado).

units establece qué tan grande será la matriz de peso en la capa. Al establecerlo en 1 aquí, estamos diciendo que habrá 1 peso para cada una de las características de entrada de los datos.

Nota: Las capas densas vienen con un término de sesgo de forma predeterminada, por lo que no necesitamos establecer useBias en true, omitiremos las llamadas posteriores a tf.layers.dense.

```
model.add(tf.layers.dense({units: 1}));
```

El código anterior crea nuestra capa de salida. Establecemos units en 1 porque queremos generar 1 número.

En este ejemplo, debido a que la capa oculta tiene 1 unidad, en realidad no necesitamos agregar la capa de salida final arriba (es decir, podríamos usar la capa oculta como la capa de salida). Sin embargo, la definición de una capa de salida separada nos permite modificar el número de unidades en la capa oculta mientras se mantiene el mapeo uno a uno de entrada y salida.

Creando una instancia

agregue el siguiente código a su función run.

```
// Create the model
const model = createModel();
tfvis.show.modelSummary({name: 'Model Summary'}, model);
```

Esto creará una instancia del modelo y mostrará un resumen de las capas en la página web.

Prepare los datos para el entrenamiento

Para obtener los beneficios de rendimiento de TensorFlow.js que hacen que los modelos de aprendizaje automático sean prácticos, necesitamos convertir nuestros datos en tensores. También realizaremos una serie de transformaciones en nuestros datos que son las mejores prácticas, a saber, barajar y normalizar.
agregue el siguiente código a su archivo script.js

```
/**
 * Convert the input data to tensors that we can use for machine
 * learning. We will also do the important best practices of _shuffling_
 * the data and _normalizing_ the data
 * MPG on the y-axis.
 */
```

```
function convertToTensor(data) {
  // envolviendo estos cálculos en el método tidy se eliminaran
  // los tensores intermedios.

  return tf.tidy(() => {
    // Step 1. aleatorizar los datos
    tf.util.shuffle(data);

    // Step 2. Convertir data a Tensor
    const inputs = data.map(d => d.horsepower)
    const labels = data.map(d => d.mpg);

    const inputTensor = tf.tensor2d(inputs, [inputs.length, 1]);
    const labelTensor = tf.tensor2d(labels, [labels.length, 1]);

    //Step 3. Normalizar los data al el rango 0 - 1 usando escala min-max
    const inputMax = inputTensor.max();
    const inputMin = inputTensor.min();
    const labelMax = labelTensor.max();
    const labelMin = labelTensor.min();

    const normalizedInputs = inputTensor.sub(inputMin).div(inputMax.sub(inputMin));
    const normalizedLabels = labelTensor.sub(labelMin).div(labelMax.sub(labelMin));

    return {
      inputs: normalizedInputs,
      labels: normalizedLabels,
      // Return the min/max bounds so we can use them later.
      inputMax,
      inputMin,
      labelMax,
      labelMin,
    }
  });
}
```

Explicamos el código enseguida:
Barajamos los datos:
// Step 1. Shuffle the data
tf.util.shuffle(data);

Aquí aleatorizamos el orden de los ejemplos que alimentaremos al algoritmo de entrenamiento. La combinación aleatoria es importante porque normalmente durante el entrenamiento el conjunto de datos se divide en subconjuntos más pequeños, llamados lotes, en los que se entrena el modelo. La combinación aleatoria ayuda a que cada lote tenga una variedad de datos de toda la distribución de datos. Al hacerlo, ayudamos al modelo:

Mejor práctica 1: siempre debe barajar sus datos antes de entregarlos a los algoritmos de entrenamiento en TensorFlow.js.

Convertir a tensores.

```
// Step 2. Convert data to Tensor
const inputs = data.map(d => d.horsepower)
const labels = data.map(d => d.mpg);

const inputTensor = tf.tensor2d(inputs, [inputs.length, 1]);
const labelTensor = tf.tensor2d(labels, [labels.length, 1]);
```

Aquí hacemos dos matrices, una para nuestros ejemplos de entrada (las entradas de caballos de fuerza) y otra para los valores de salida verdaderos (que se conocen como etiquetas en el aprendizaje automático).

Luego convertimos cada dato de matriz a un tensor 2D. El tensor tendrá una forma de [num_examples, num_caracteristicas_por_example]. Aquí tenemos inputs.length ejemplos y cada ejemplo tiene 1 característica de entrada (los caballos de fuerza).

Normalizar los datos

```
//Step 3. Normalize the data to the range 0 - 1 using min-max scaling
const inputMax = inputTensor.max();
const inputMin = inputTensor.min();
const labelMax = labelTensor.max();
const labelMin = labelTensor.min();

const normalizedInputs = inputTensor.sub(inputMin).div(inputMax.sub(inputMin));
const normalizedLabels = labelTensor.sub(labelMin).div(labelMax.sub(labelMin));
```

A continuación, hacemos otra práctica recomendada para la capacitación en aprendizaje automático. Normalizamos los datos. Aquí normalizamos los datos en el rango numérico 0-1 usando la escala min-max. La normalización es importante porque los componentes internos de muchos modelos de aprendizaje automático que construirá con tensorflow.js están diseñados para trabajar con números que no son demasiado grandes. Rangos comunes para normalizar datos para incluir 0 a 1 o -1 a 1. Tendrá más éxito entrenando a sus modelos si se acostumbra a normalizar sus datos a un rango razonable.

Mejor práctica 2: siempre debe considerar normalizar sus datos antes de entrenar. Algunos conjuntos de datos se pueden aprender sin normalización, pero la normalización de sus datos a menudo eliminará toda una clase de problemas que evitarían un aprendizaje efectivo.

Puede normalizar sus datos antes de convertirlos en tensores. Lo hacemos después porque podemos aprovechar la vectorización en TensorFlow.js para realizar las operaciones de escalado min-max sin escribir ningún explícito para los bucles.

Devuelve los datos y los límites de normalización.

```
return {
  inputs: normalizedInputs,
  labels: normalizedLabels,
  // Return the min/max bounds so we can use them later.
```

```
  inputMax,
  inputMin,
  labelMax,
  labelMin,
}
```

Queremos mantener los valores que usamos para la normalización durante el entrenamiento para que podamos normalizar las salidas para volver a nuestra escala original y permitirnos normalizar los datos de entrada futuros de la misma manera.

Entrenar el modelo

Con nuestra instancia de modelo creada y nuestros datos representados como tensores, tenemos todo listo para comenzar el proceso de capacitación.

```
async function trainModel(model, inputs, labels) {
  // Prepare the model for training.
  model.compile({
    optimizer: tf.train.adam(),
    loss: tf.losses.meanSquaredError,
    metrics: ['mse'],
  });

  const batchSize = 32;
  const epochs = 50;

  return await model.fit(inputs, labels, {
    batchSize,
    epochs,
    shuffle: true,
    callbacks: tfvis.show.fitCallbacks(
      { name: 'Training Performance' },
      ['loss', 'mse'],
      { height: 200, callbacks: ['onEpochEnd'] }
    )
  });
}
```

Analicemos el código:

Preparándose para el entrenamiento

```
// Prepare the model for training.
model.compile({
  optimizer: tf.train.adam(),
  loss: tf.losses.meanSquaredError,
  metrics: ['mse'],
});
```

Tenemos que "compilar" el modelo antes de entrenarlo. Para hacerlo, tenemos que especificar una serie de cosas muy importantes:

- optimizador: Este es el algoritmo que regirá las actualizaciones del modelo, ya que ve ejemplos. Hay muchos optimizadores disponibles en TensorFlow.js. Aquí hemos elegido el optimizador adam, ya que es bastante efectivo en la práctica y no requiere configuración.
- pérdida: esta es una función que le dirá al modelo qué tan bien le está yendo al aprender cada uno de los lotes (subconjuntos de datos) que se muestra. Aquí usamos meanSquaredError para comparar las predicciones hechas por el modelo con los valores verdaderos.

```
const batchSize = 32;
const epochs = 50;
```

A continuación, elegimos un batchSize y varias épocas:

- batchSize se refiere al tamaño de los subconjuntos de datos que el modelo verá en cada iteración de entrenamiento. Los tamaños de lote comunes tienden a estar en el rango de 32-512. Realmente no hay un tamaño de lote ideal para todos los problemas y está más allá del alcance de este tutorial para describir las motivaciones matemáticas para varios tamaños de lote.
- épocas se refiere a la cantidad de veces que el modelo mirará todo el conjunto de datos que usted le proporcione. Aquí tomaremos 50 iteraciones a través del conjunto de datos.

Iniciar el loop de entrenamiento

```
return await model.fit(inputs, labels, {
  batchSize,
  epochs,
  callbacks: tfvis.show.fitCallbacks(
    { name: 'Training Performance' },
    ['loss', 'mse'],
    { height: 200, callbacks: ['onEpochEnd'] }
  )
});
```

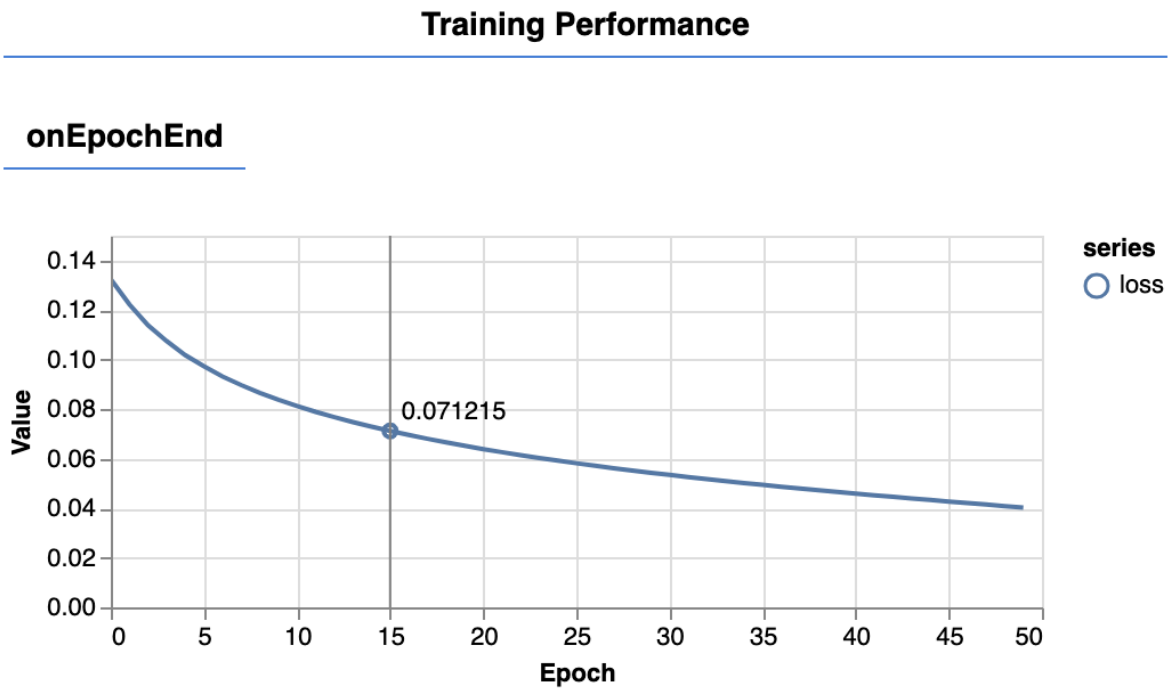
model.fit es la función que llamamos para iniciar el ciclo de entrenamiento. Es una función asincrónica, por lo que le devolvemos la promesa que nos brinda para que la persona que llama pueda determinar cuándo se completa el entrenamiento.
Para monitorear el progreso de la capacitación, pasamos una función callback a model.fit. Usamos tfvis.show.fitCallbacks para generar funciones que tracen gráficos para la métrica "pérdida" y "mse" que especificamos anteriormente.

Todo junto
Ahora tenemos que llamar a las funciones que hemos definido desde nuestra función de ejecución.
Agregue el siguiente código al final de su función run.

```
// Convert the data to a form we can use for training.
const tensorData = convertToTensor(data);
const {inputs, labels} = tensorData;

// Train the model
await trainModel(model, inputs, labels);
console.log('Done Training');
```

Cuando actualice la página, después de unos segundos, debería ver la actualización de los siguientes gráficos.



Estos son creados por las callback que creamos anteriormente. Muestran la pérdida y la mse, promediadas en todo el conjunto de datos, al final de cada época.

Cuando entrenamos a un modelo, queremos ver cómo baja la pérdida. En este caso, debido a que nuestra métrica es una medida de error, también queremos que disminuya.

Si quieres entender lo que sucede debajo del capó durante el entrenamiento. Lee nuestra guía o mira esto: https://www.youtube.com/watch?v=IHZwWFHWa-w

Hacer predicciones

Ahora que nuestro modelo está entrenado, queremos hacer algunas predicciones. Vamos a evaluar el modelo al ver lo que predice para un rango uniforme de números de horsepower bajas a altas.

Agregue la siguiente función a su archivo script.js

```
function testModel(model, inputData, normalizationData) {
  const {inputMax, inputMin, labelMin, labelMax} = normalizationData;

  // Generate predictions for a uniform range of numbers between 0 and 1;
  // We un-normalize the data by doing the inverse of the min-max scaling
  // that we did earlier.
  const [xs, preds] = tf.tidy(() => {

    const xs = tf.linspace(0, 1, 100);
    const preds = model.predict(xs.reshape([100, 1]));

    const unNormXs = xs
      .mul(inputMax.sub(inputMin))
      .add(inputMin);

    const unNormPreds = preds
      .mul(labelMax.sub(labelMin))
      .add(labelMin);

    // Un-normalize the data
    return [unNormXs.dataSync(), unNormPreds.dataSync()];
  });


  const predictedPoints = Array.from(xs).map((val, i) => {
    return {x: val, y: preds[i]}
  });

  const originalPoints = inputData.map(d => ({
    x: d.horsepower, y: d.mpg,
  }));


  tfvis.render.scatterplot(
    {name: 'Model Predictions vs Original Data'},
    {values: [originalPoints, predictedPoints], series: ['original', 'predicted']},
    {
      xLabel: 'Horsepower',
      yLabel: 'MPG',
      height: 300
    }
  );
}
```

Algunas cosas para notar en la función anterior.

```
const xs = tf.linspace(0, 1, 100);
const preds = model.predict(xs.reshape([100, 1]));
```

Generamos 100 nuevos ‘ejemplos' para alimentar el modelo. Model.predict es cómo introducimos esos ejemplos en el modelo. Tenga en cuenta que deben tener una forma similar ([num_examples, num_features_per_example]) como cuando hicimos entrenamiento.

```
// Un-normalize the data
const unNormXs = xs
  .mul(inputMax.sub(inputMin))
  .add(inputMin);

const unNormPreds = preds
  .mul(labelMax.sub(labelMin))
  .add(labelMin);
```

Para que los datos vuelvan a nuestro rango original (en lugar de 0-1), usamos los valores que calculamos mientras normalizamos, pero solo invertimos las operaciones.

```
return [unNormXs.dataSync(), unNormPreds.dataSync()];
```

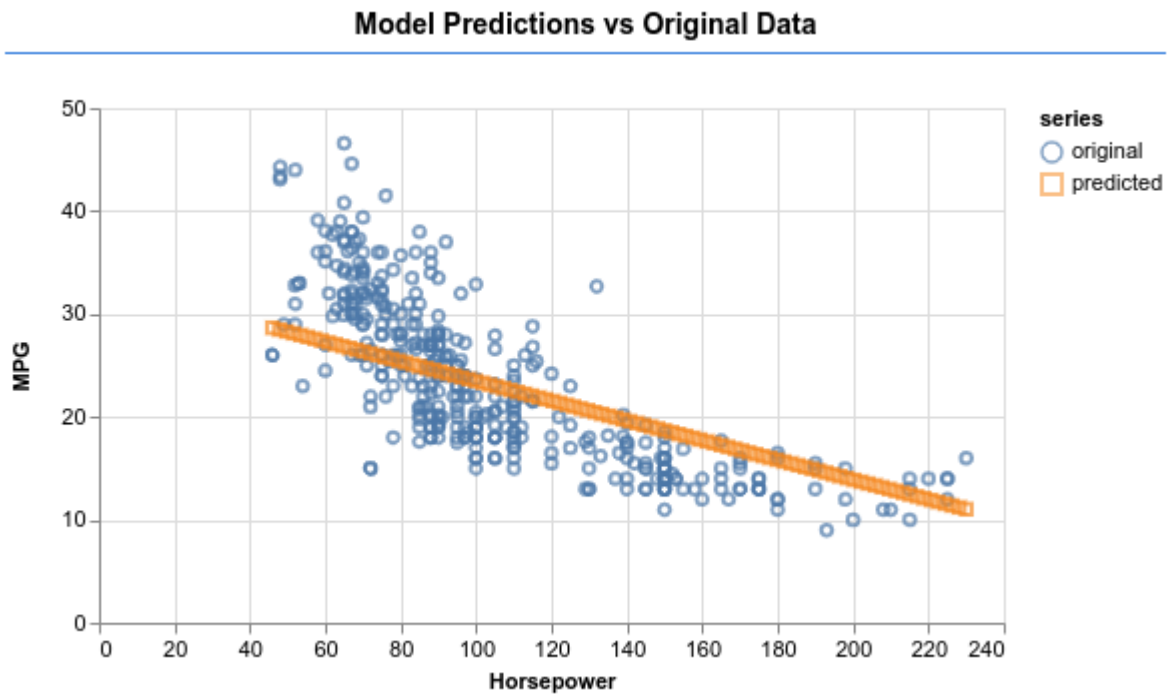
dataSync() es un método que podemos usar para obtener una matriz de tipos de los valores almacenados en un tensor. Esto nos permite procesar esos valores en JavaScript normal. Esta es una versión síncrona de .data() método que generalmente se prefiere.

Finalmente, usamos tfjs-vis para trazar los datos originales y las predicciones del modelo.

Agregue el siguiente código a su función run.

```
// Make some predictions using the model and compare them to the
// original data
testModel(model, data, tensorData);
```

Actualice la página y debería ver algo como lo siguiente una vez que el modelo termine de entrenar.



Acaba de entrenar un modelo simple de aprendizaje automático. Actualmente realiza lo que se conoce como regresión lineal que intenta ajustar una línea a la tendencia presente en los datos de entrada.

Los pasos para entrenar un modelo de aprendizaje automático incluyen:

Formula tu tarea:

- ¿Es un problema de regresión o uno de clasificación?
- ¿Se puede hacer esto con aprendizaje supervisado o aprendizaje no supervisado?
- ¿Cuál es la forma de los datos de entrada? ¿Cómo deberían ser los datos de salida?

Prepara tus datos:

- Limpie sus datos e inspeccione manualmente los patrones cuando sea posible
- Mezcle sus datos antes de usarlos para el entrenamiento
- Normalice sus datos en un rango razonable para la red neuronal. Por lo general, 0-1 o -1-1 son buenos rangos para datos numéricos.
- Convierte tus datos en tensores

Construye y ejecuta tu modelo:

- Defina su modelo usando tf.sequential o tf.model y luego agregue capas usando tf.layers. *
- Elija un optimizador (adam suele ser bueno) y parámetros como el tamaño del lote y el número de épocas.
- Elija una función de pérdida adecuada para su problema y una métrica de precisión para ayudarlo a evaluar el progreso. meanSquaredError es una función de pérdida común para problemas de regresión.

- Monitoree el entrenamiento para ver si la pérdida está bajando

Evalúa tu modelo

- Elija una métrica de evaluación para su modelo que pueda monitorear mientras entrena. Una vez que esté entrenado, intente hacer algunas predicciones de prueba para tener una idea de la calidad de la predicción.

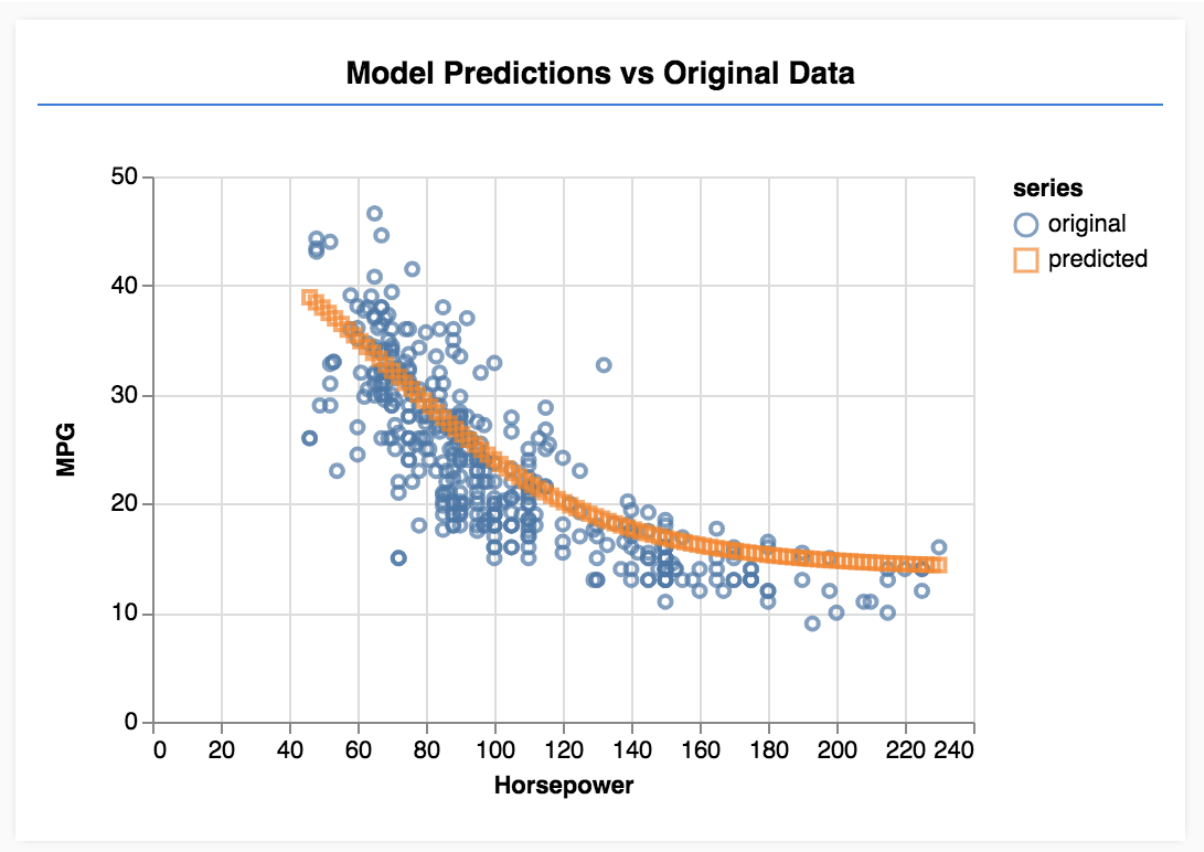
Cosas para probar

- Experimenta cambiando el número de épocas. ¿Cuántas épocas necesitas antes de que el gráfico se ajuste?
- Experimente aumentando el número de unidades en la capa oculta.
- Experimente agregando más capas ocultas entre la primera capa oculta que agregamos y la capa de salida final. El código para estas capas adicionales debería verse más o menos así.

```
model.add(tf.layers.dense({units: 50, activation: 'sigmoid'}));
```

La novedad más importante sobre estas capas ocultas es que introducen una función de activación no lineal, en este caso, la activación sigmoidea. Para obtener más información sobre las funciones de activación, consulte este artículo: https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/anatomy .

Vea si puede lograr que el modelo produzca resultados como en la imagen a continuación.



Referencias

Esta es una traducción de: AA: https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss?hl=es-419
Machine learning:
https://developers.google.com/machine-learning/crash-course/logistic-regression/calculating-a-probability

tensorflow:
https://www.tensorflow.org/js

Resumen redes neuronales y TF en python
https://relopezbriega.github.io/blog/2016/06/05/tensorflow-y-redes-neuronales/

Ejemplo en JS
http://www.jortilles.com/tensorflow-js/

Jugando con redes neuronales:
https://playground.tensorflow.org