Práctica 1: Búsqueda Iterativa de Óptimos y Regresión Lineal

Alejandro Alonso Membrilla

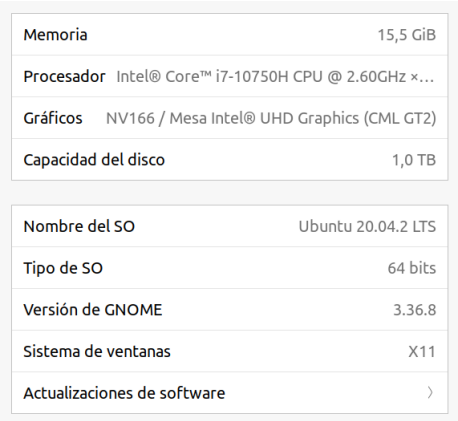
Contents

Introducción

Esta práctica consistirá en una serie de experimentos relacionados con métodos de optimización usadas ampliamente en el Aprendizaje Automático, tanto en un contexto general como en el de la clasificación de imágenes de números mediante técnicas de regresión lineal. Veremos el gradiente descendiente, tanto en su versión determinista como estocástica, además del método de la pseudoinversa para el caso de la regresión lineal.

Procuraremos dar contexto a cada experimento y sentido a los resultados, además de usar estos para analizar la eficacia de los algoritmos de optimización utilizados.

El código implementado para los experimentos ha sido escrito en Python, haciendo uso de la biblioteca `numpy` para álgebra lineal y vectorización. Los cálculos y los gráficos obtenidos para muchos experimentos en esta práctica han sido realizados en el ordenador personal del autor de la misma, Alejandro Alonso Membrilla, que cuenta con las siguientes características:



Otras bibliotecas utilizadas durante los ejercicios se indican en el código y en el apartado de referencias.

Optimización Iterativa: Gradiente Descendiente

Para la primera sección de esta práctica, hemos implementado el algoritmo de Gradiente Descendiente, o *Batch Gradient Descent*, siguiendo las descripción facilitada en la presentación de la práctica, para ser aplicado a una función diferenciable. Tanto dicha función como su derivada (gradiente) han de ser pasadas como parámetro a la función, junto un punto inicial para la búsqueda iterativa y un ratio de aprendizaje. La implementación del algoritmo se encuentra en el script adjunto a la presente memoria.

Hemos experimentado con los parámetros del gradiente descendiente a partir de dos funciones distintas a minimizar, que vemos en las siguientes subsecciones.

Primera función objetivo: $E(u,v) = (u^3e^{(v-2)} - 2v^2e^{-u})^2$

Empezaremos mostrando la expresión analítica del gradiente de esta función:

$$\nabla E = (\frac{\partial E}{\partial u}, \frac{\partial E}{\partial v})$$

Donde

$$\frac{\partial E}{\partial u} = 2(u^3e^{(v-2)} - 2v^2e^{-u})(3u^2ev - 2 + 2v^2e-u)$$

Y

$$\frac{\partial E}{\partial v} = 2(u^3 e^{(v-2)} - 2v^2 e^{-u})(u^3 ev - 2 - 4ve{-}u)$$

A continuación, arrancamos el algoritmo con los parámetros pedidos de $(u, v) = (1, 1)$ y $\eta = 0.1$.

Resultados: el algoritmo tarda 10 iteraciones en obtener un resultado aproximado de $3.11396 \times 10^{-15} < 10^{-14}$. Las coordenadas encontradas para las que la función E toma dichos valor son $(u, v) \approx (1.15729, 0.91084)$.

Segunda función objetivo: $f(x, y) = (x + 2)^2 + 2(y - 2)^2 + 2\sin(2\pi x)\sin(2\pi y)$

De nuevo, calculamos la expresión analítica del gradiente de la función a minimizar. En este caso:

$$\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$$

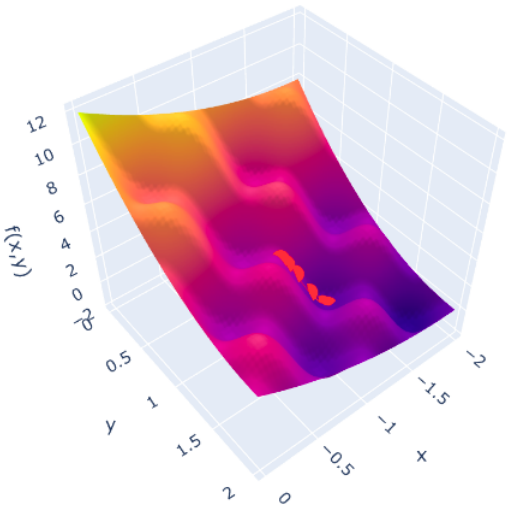
Donde

$$\frac{\partial f}{\partial x} = 2(x + 2) + 4\pi \cos(2\pi x)\sin(2\pi y)$$

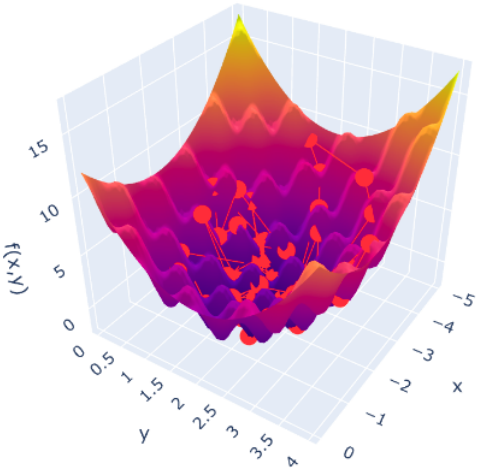
Y

$$\frac{\partial f}{\partial y} = 4(y - 2) + 4\pi \sin(2\pi x)\cos(2\pi y)$$

La función anterior ha sido minimizada usando gradiente descendiente para el punto inicial $(x_0 = -1, y_0 = 1)$, con un $\eta = 0.01$ y un número máximo de iteraciones de 50. El "camino" recorrido por el algoritmo en el espacio de soluciones puede verse en la siguiente gráfica 3D, creada usando la API de Plotly, una biblioteca de Python para visualización de datos estadísticos.

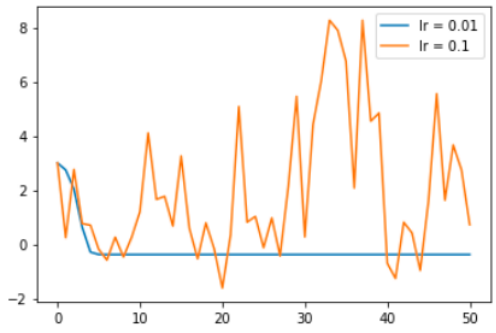


Observamos que el valor de la función decrece en cada paso, hasta detenerse en un óptimo local cercano siguiendo las indicaciones dadas por el gradiente. Repetimos el experimento para $\eta = 0.1$, encontrándonos con el siguiente resultado:



Haciendo un *zoom out* a la superficie definida por la gráfica de la función objetivo, vemos que esta tiene una forma relativamente compleja, llena de "montañas" y de "valles" (extremos locales), aunque se observa una tendencia general de la función a decrecer al acercarnos al entorno del punto $(x, y) = (-2, 2)$.

Además, vemos claramente que al aumentar el ratio de aprendizaje de 0.01 a 0.1, la función empieza a comportarse de forma errática. La dirección dada por el gradiente ya no es de utilidad, puesto que la longitud del salto dado entre una iteración y la siguiente es demasiado alta, haciendo que el descenso de gradiente se salte el óptimo local. A continuación, vemos en una gráfica una comparación entre los valores tomados por la función f a lo largo del proceso de optimización para ambos valores de η:



A priori, podríamos considerar que el comportamiento del descenso de gradiente parece mejor cuando η es menor. Sin embargo, observando la gráfica anterior notamos que, para algunas coordenadas encontradas a lo largo del proceso iterativo, la función de pérdida alcanza valores más bajos para $\eta = 0.1$ que para $\eta = 0.01$. Esto quiere decir que si modificásemos nuestra implementación del Descenso de Gradiente para conservar la mejor solución encontrada hasta el momento (*Pocket Algorithm*), la solución obtenida para $\eta = 0.1$ hubiese sido la mejor de las dos.

Por último, mostraremos una tabla con los resultados encontrados por el algoritmo de Gradiente Descendiente para distintas coordenadas iniciales. El *learning rate* utilizado será 0.01, al ser el que ha dado resultados más estables para la implementación realizada (sin memoria de la mejor solución):

Punto inicial	**Óptimo encontrado**	**f**
(-0.5, -0.5)	(-0.7935, -0.1260)	9.1251
(1.0,1.0)	(0.6774, 1.2905)	6.4376
(2.1,-2.1)	(0.1488, -0.0961)	12.4910
(-3.0,3.0)	(-2.7309, 2.7133)	-0.3812
(-2.0,2.0)	(-2.0000, 2.0000)	-0.0000

En lo que respecta a la tabla anterior, a simple vista cabe destacar que ninguna de las soluciones obtenidas coinciden entre sí. La mayor parte son relativamente cercanas, en términos de distancia, al punto inicial.

Además, vemos como el valor de f difiere notablemente entre las distintas soluciones. Dados los datos de la última tabla, hemos calculado una desviación típica (por coordenadas) entre las soluciones encontradas de $(1.27704, 1.12842)$, muy a tener en cuenta considerando que todos los puntos iniciales se han tomado en el rectángulo $[-3, 2.1] \times [-2.1, 3]$. Además, los valores de f para estas soluciones presentan una relativamente elevada desviación típica de $5,054$, acorde a las observaciones realizadas a simple vista.

Conclusiones del apartado 1

En lo que respecta a calcular el óptimo global de una función arbitraria, concluimos que el método de gradiente descendiente no es suficiente.

Detengámonos en el caso de una función convexa, para la cual todo óptimo local es un óptimo global. Como hemos visto en los experimentos anteriores, y como se deduce de la construcción del método de gradiente descendiente, este algoritmo debe acercarnos, en teoría, al óptimo local más cercano, que en el caso de una función convexa es un óptimo global. Sin embargo, al tratarse de una función convexa arbitraria, no podemos fijar un único ratio de aprendizaje (longitud de salto) que, como hemos visto, ha de ser suficientemente pequeño como para que el descenso de gradiente nos acerque a la solución sin "saltársela". Si decidimos fijar un ratio de aprendizaje muy pequeño que nos asegure acercarnos lo máximo posible a los óptimos de un mayor número de funciones convexas, tendremos que pagar el precio de hacer el algoritmo menos eficiente, en términos de velocidad de ejecución, para otras funciones convexas más sencillas de optimizar.

Si consideramos ahora todo tipo de funciones, no solo convexas, nos encontramos con el problema visto para la función $f(x, y)$ anterior. Para un ratio de aprendizaje muy elevado no tenemos garantías sobre la convergencia del algoritmo, mientras que para un ratio bajo caeremos en algún óptimo local, posiblemente distinto al global. Dado que queremos calcular el óptimo global de una función arbitraria, ninguna de estas opciones es razonable.

A esto se suma el problema de la elección del punto inicial. En una función no convexa, el punto inicial determinará cual es el óptimo local más cercano. Elegir un punto inicial cercano al global y un ratio de aprendizaje bajo nos asegurará converger al óptimo global, pero no es posible encontrar un valor para estos parámetros que nos asegure dicha convergencia para todas las funciones de pérdida.

Estos problemas no se encuentran exclusivamente en el Gradiente Descediente, sino en todos los algoritmos de optimización por computadora. De acuerdo con los teoremas de *No Hay Barra Libre* o *No Free Lunch*, dado un par de algoritmos de optimización deterministas cualesquiera, un dominio X y un codominio Y finitos, el resultado promedio de aplicar ambos algoritmos sobre todo el conjunto de funciones de X en Y es el mismo, independientemente del número de iteraciones que se le den a ambos. Nótese que las condiciones anteriores no son restrictivas, puesto que todos los algoritmos implementables en una computadora moderna son deterministas (incluso los pseudoaleatorios), y todas las funciones deben restringirse al conjunto de números representables en código máquina (dominio y codominio finitos).

En conclusión, no es posible encontrar un algoritmo que sea "el mejor" para optimizar globalmente una función arbitraria, puesto que todos los algoritmos, en promedio, rendirán igual que una búsqueda aleatoria, o una por fuerza bruta de la solución óptima.

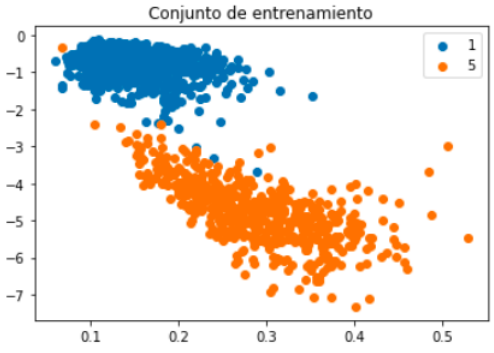
Regresión Lineal

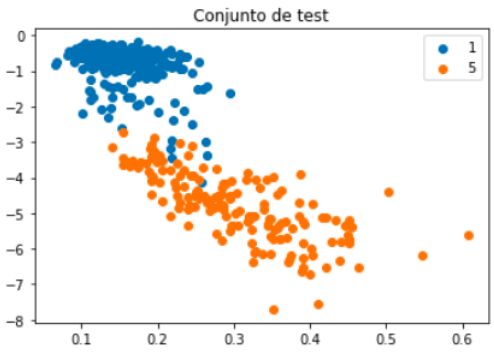
Este ejercicio consistirá en el uso de técnicas de regresión lineal para clasificación. En un primer apartado trataremos la clasificación de dígitos manuscritos en función de dos características: su simetría respecto al eje vertical y su intensidad media. En el segundo apartado generaremos una muestra aleatoria siguiendo una serie de patrones y separaremos las distintas clases generadas usando características lineales y no lineales.

Dígitos manuscritos

Una vez cargado el dataset, desde los ficheros *X_train, y_train, X_test* e *y_test*, eliminamos todas las instancias que no correspondan a 1s y 5s, quedándonos con un total de 1561 instancias de entrenamiento y 424 de test. A continuación mostramos en un gráfico las características de cada elemento, junto con la clase a la

que pertenecen, tanto en el conjunto de entrenamiento como en el de test:





El problema a resolver consiste en encontrar una configuración de pesos, w, cumpliendo que estos hagan mínimo el *riesgo empírico*, o error cuadrático medio en la muestra de entrenamiento, dado por la siguiente fórmula:

$$E_{in}(w) = \frac{1}{N}\sum_{i=1}^{N}(w^T x_i - y_i)^2$$

donde w es el vector de pesos, x_i es un vector de la forma $(1, x_i1, ..., x_id)$, con $x_{i1}, ..., x_{id}$ las características medidas para cada instancia del conjunto de datos (en este caso $d = 2$), e $y \in \{-1, 1\}$ representa la etiqueta asignada a cada elemento, 1 si el dígito manuscrito es un 1 y -1 si es un 5. Nuestro objetivo es que los pesos calculados a partir del conjunto de entrenamiento sirvan para separar una muestra no observada previamente, el conjunto de test, el cual sigue, asumiblemente, la misma distribución de probabilidad que la de la muestra usada para el aprendizaje de los pesos.

Empezaremos mostrando los resultados obtenidos aplicando el método de la **pseudoinversa**: siguiendo la explicación dada en teoría, hemos llamado $X \in M_{N\times(1+d)}$ a la matriz formada por los vectores x_i definidos en el párrafo anterior, hemos calculado la matriz pseudoinversa de X como

$$X^\dagger = (X^T X)^{-1} X^T$$

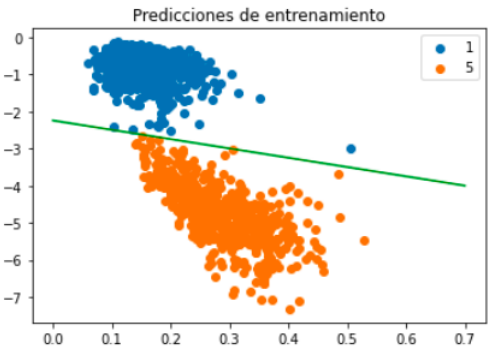
y, finalmente,

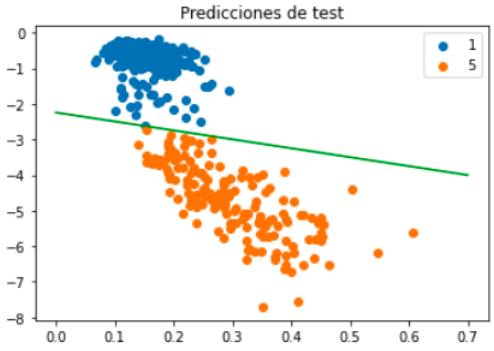
$$w = X^\dagger y$$

nos da el valor de w que minimiza el riesgo empírico. A continuación se muestran los errores cuadráticos y de clasificación (EC, proporción de instancias mal clasificadas) obtenidos para los pesos determinados, tanto para el conjunto de entrenamiento como para el de test:

$E_{in} = 0.079187$ $\quad$ $E_{out} = 0.13095$ $\quad$ $EC_{in} = 0.00512$ $\quad$ $EC_{out} = 0.01651$

Vemos un error de clasificación en torno al 0.5% en entrenamiento y al 1% en test. A continuación, visualizamos el modelo de clasificación resultante mediante una línea que separa ambas clases:





A continuación intentaremos resolver el mismo problema usando Descenso de Gradiente Estocástico. Este algoritmo no nos garantiza minimizar óptimamente el riesgo empírico, pero es aplicable en muchos otros problemas distintos a la regresión lineal.

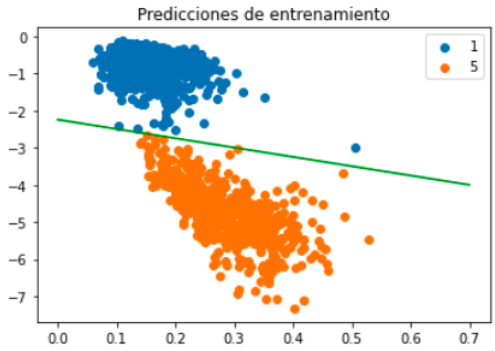
El ratio de aprendizaje establecido inicialmente (y el único, como se justificará posteriormente) en este caso es de 0.01, mientras que el tamaño de *minibatch* se ha fijado a 32 siguiendo las sugerencias dadas en este artículo. El número de épocas dadas para el entrenamiento ha sido de 10, considerando que $10\lceil\frac{1561}{32}\rceil = 490$ iteraciones son suficientes para un problema de esta magnitud (solamente 3 pesos). El valor de w se inicializa como tres escalares aleatorios tomados a partir de una distribución uniforme en el intervalo $[0, 1)$ usando el módulo `random` de `numpy`, aunque cualquier valor no muy elevado debería permitir al SGD acercarse al óptimo en no muchas iteraciones.

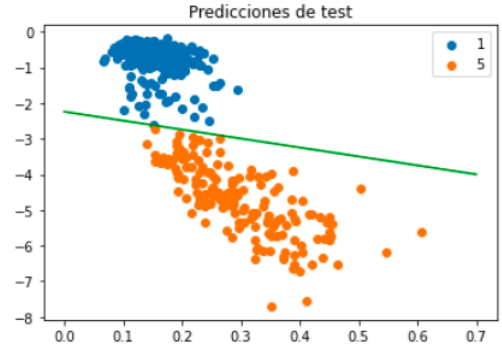
A modo de anotación, no se ha considerado aplicar una normalización del conjunto de datos al aplicar SGD en este caso, puesto que esto no debería afectar a la solución, como se indica en esta respuesta en StackExchange.

Los resultados obtenidos usando SGD son los siguientes:

$E_{in} = 0.08107$ $E_{out} = 0.13665$ $EC_{in} = 0.00512$ $EC_{out} = 0.01651$

Mientras que la separación resultante está representada en la siguiente gráfica:





Podemos ver una comparación entre SGD y la pseudoinversa en la siguiente tabla:

Error	**Pseudoinversa**	**SGD**
E_{in}	0.079187	0.08042
E_{out}	0.13095	0.13461
EC_{in}	0.00512	0.00512
EC_{out}	0.01651	0.01651

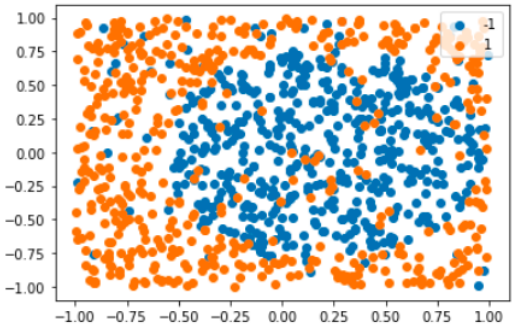
Si bien el error cuadrático medio es ligeramente mayor al usar SGD tanto en entrenamiento como en test, estos son muy cercanos a los obtenidos por la pseudoinversa. Además, el error de clasificación coincide para ambos algoritmos. Aunque solamente hemos repetido el experimento una vez y no podemos afirmar que nuestros resultados sean estadísticamente significativos, parece que los parámetros que hemos escogido en un principio para el SGD funcionan adecuadamente para este conjunto de datos, y que este algoritmo es capaz de encontrar buenas soluciones para este problema.

Modelos lineales y no lineales

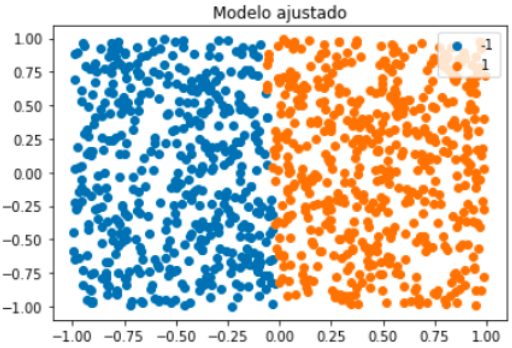
El conjunto de datos a tratar en este experimento será generado aleatoriamente siguiendo una serie de reglas:

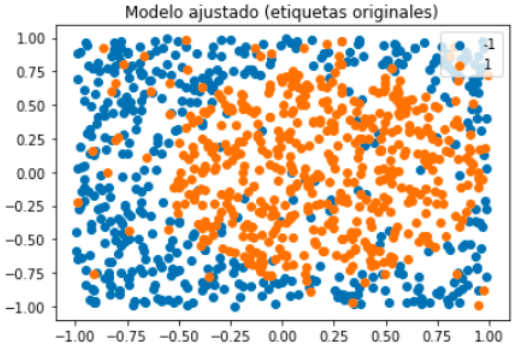
1. La muestra serán 1000 puntos muestreados uniformemente en el cuadrado $[-1, 1] \times [-1, 1]$.
2. La etiqueta de cada punto $x = (x_1, x_2)$ será inicialmente elegida como la función $f(x) = signo((x_1 - 0.2)^2 + x_2^2 - 0.6)$.
3. El 10% de las intancias, elegidas de forma aleatoria, cambiará su etiqueta por la opuesta.

En un primer momento, vemos que la función f incorpora términos cuadráticos. Representando los distintos elementos del dataset generado coloreados por clases, obtenemos el siguiente gráfico:



Ahora aplicaremos el algoritmo de Gradiente Descendiente Estocástico igual que hemos hecho en el apartado anterior, y con los mismos parámetros. El resultado puede observarse en las siguientes gráficas:





Vemos como la línea que separa ambas clases queda bastante centrada, y asigna al lado derecho, aquel con más puntos naranjas, el color naranja, mientras que al lado con más puntos azules le asigna el color azul.

Los resultados numéricos son tan malos como cabría esperar a partir de la visualización anterior.

$$E_{in} = 0.91814 \qquad EC_{in} = 0.35700$$

Al repetir el experimento completo 1000 veces, añadiendo un conjunto test generado de la misma forma que el generado para aprendizaje, obtenemos los siguientes resultados promedio:

$$E_{in} = 0.93259 \qquad E_{out} = 0.93804 \qquad EC_{in} = 0.40453 \qquad EC_{out} = 0.407985$$

Los errores cuadráticos medios E_{in} y E_{out} no pueden usarse, sin otra referencia, para determinar la bondad del algoritmo. Sin embargo, el error de clasificación sí que puede ser un indicador por sí solo de la utilidad de nuestro clasificador. Un error muy cercano al 50% hace que el modelo entrenado no sea muy superior a un "etiquetado aleatorio". En este caso, los resultados promedio son aun peores que los vistos anteriormente, lo

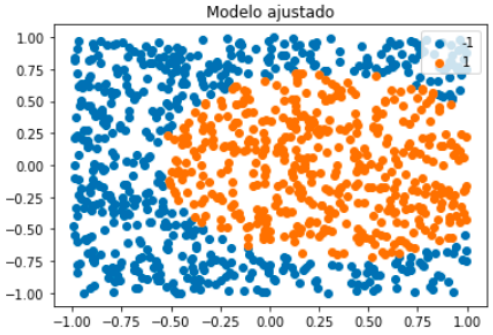
cual refuerza la tesis de que una separación del dataset usando características lineales no parece adecuada para clasificar instancias etiquetadas mediante una función no lineal como es f.

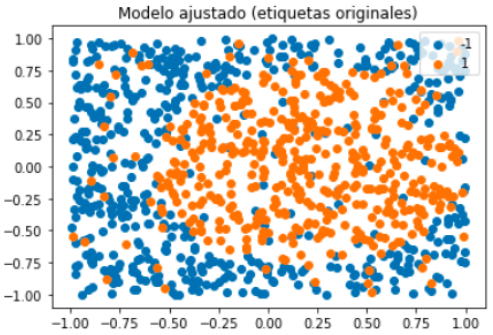
A continuación, repetiremos el mismo experimento llevado a cabo anteriormente (generación aleatoria de la muestra y de sus etiquetas a partir de f y cálculo del error de clasificación) pero en esta ocasión calcularemos previamente una serie de características cuadráticas para cada elemento del dataset, que usaremos en el proceso de optimización con SGD de la misma forma que hemos usado las características lineales anteriormente. De esta forma, nuestro modelo podrá aprender a aproximar funciones no lineales y a hacer separaciones no lineales del dataset.

Los errores obtenidos por el método extendido a características cuadráticas son:

$$E_{in} = 0.60121 \qquad EC_{in} = 0.14600$$

y la comparación entre la clasificación resultante y las etiquetas originales queda como sigue:





Observamos que esta vez la función aproximada por nuestro clasificador tiene una forma mucho más parecida a la función real, aunque en este caso la elipse formada por los puntos naranjas queda algo trasladada (respecto a la original) y, razonablemente, nuestro modelo no presenta ruido como sí lo hace en el conjunto original.

Los resultados promedio al repetir el experimento 1000 veces son:

$$E_{in} = 0.68424 \qquad E_{out} = 0.68846 \qquad EC_{in} = 0.17629 \qquad EC_{out} = 0.17977$$

Teniendo en cuenta que el 10% de los puntos presenta ruido aleatorio y no obedecen al patrón general de la muestra, una precisión de clasificación en torno al 82% es una gran mejoría respecto del modelo original.

Adicionalmente, experimentaremos con los efectos de cambiar el tamaño de *minibatch* al aplicar SGD con las características no lineales. Los resultados promedios de repetir el experimento 1000 veces con cada tamaño de *minibatch* pueden verse en la siguiente tabla:

Batch size	**Ein**	**Eout**	**ECin**	**ECout**
256	1.17525	1.17798	0.47589	0.477281
128	0.95656	0.960302	0.40047	0.40313
64	0.79911	0.80336	0.26946	0.27200
32	0.68652	0.69100	0.17839	0.18147
16	0.61182	0.61763	0.13934	0.14227
8	0.57714	0.58370	0.13622	0.13903

Vemos una tendencia en los resultados a mejorar con la reducción del tamaño de *minibatch*.

Cabe destacar el hecho de que un modelo con características lineales y cuadráticas como el diseñado en este ejercicio incluye al modelo de características únicamente lineales, en el sentido de que puede aproximar las mismas funciones que el modelo lineal simplemente fijando los pesos correspondientes a las características no lineales a 0. Por esto, siempre cabría esperar que el primer modelo funcionase mejor que el segundo, al menos en teoría. Las causas por las que esto podría no suceder en la práctica son el aumento de la dimensionalidad del espacio de pesos/parámetros de optimización, y su consecuente incremento en la dificultad para ajustar el

modelo entrenado. Esto lleva al efecto que observamos en la tabla anterior, correspondiente al experimento con los tamaños de *minibatch*, en la que vemos que usar lotes muy grandes conlleva peores resultados, probablemente debido a que se reduce la variabilidad estocástica del algoritmo y se promueve la caída en óptimos locales. Sin embargo, en casos como el que nos ocupa con un número de parámetros tan reducido, el ajuste de hiperparámetros y el entrenamiento del modelo más complejo siguen siendo suficientemente sencillos como para que merezca la pena su uso.

A vista de los resultados, concluimos que para el problema afrontado en este apartado el ajuste de un modelo cuadrático es más adecuado que el de un modelo lineal.

BONUS: Método de Newton

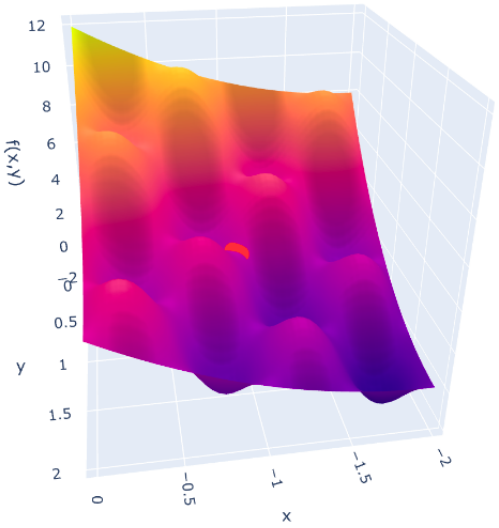
En adición al estudio realizado en la primera sección sobre Gradiente Descendiente, veremos el método relajado de Newton como técnica complementaria de optimización iterativa, en la que el vector de pesos w se modifica siguiendo la siguiente regla:

$$\Delta w = -\eta H_f(w)^{-1} \nabla f(w_o)$$

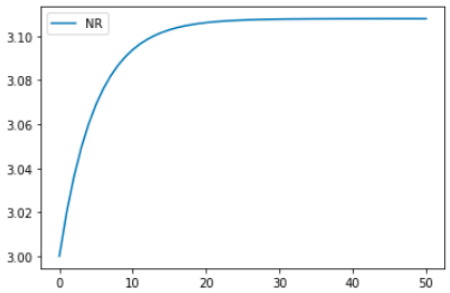
donde w_o es la solución actual en una iteración determinada, $\eta \in \mathbb{R}^+$ es un escalar análogo al ratio de aprendizaje en GD, f es la función a optimizar y H_f es la matriz hessiana de f, esto es, la matriz jacobiana de ∇f, o la generalización de la segunda derivada para funciones multivariables, dada por la siguiente expresión (en el caso de dos variables):

$$H_f(x,y) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2}(x,y) & \frac{\partial^2 f}{\partial xy}(x,y) \\ \frac{\partial^2 f}{\partial xy}(x,y) & \frac{\partial^2 f}{\partial y^2}(x,y) \end{pmatrix}$$

Al repetir el experimento realizado para la función f en la sección sobre Gradiente Descendiente, usando un ratio de aprendizaje $\eta = 0.1$ y para un punto inicial $(-1.0, 1.0)$, nos encontramos con el resultado siguiente:



En particular, la solución encontrada es $(x, y) = (-0.94633, 0.97171)$ y $f(x, y) = 3.10798$. La evolución de f a lo largo del proceso iterativo es la siguiente:



Claramente, el algoritmo no parece buscar un mínimo local, sino que parece acercarse a un punto de silla de la función. Esto se debe a que el proceso anterior surge como generalización a varias variables del método de Newton-Raphson para resolución de ecuaciones:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

El motivo por el que es importante destacar esto es que, al aplicar el método de Newton clásico sobre la derivada de una función, lo que hacemos es buscar un punto donde dicha derivada se anula. Esto quiere decir que el algoritmo tenderá hacia un punto crítico de cualquier tipo, no solamente mínimos. Para saber si nos estamos acercando a un máximo o a un mínimo, tenemos que fijarnos en los valores propios de la hessiana (al ser simétrica, la hessiana será siempre diagonalizable). Cuando todos los valores propios sean positivos, la función f será convexa, si todos son negativos será cóncava y si hay de ambos tipos no será de ninguno de los dos (nos encontraremos cerca de un punto de silla). En todo esto, asumimos que la matriz hessiana es no singular.

En el caso de funciones de dos dimensiones, podemos reducir este problema a mirar el determinante y la traza de $H_f(x, y)$ (recordemos que tanto el determinante como la traza son invariantes entre matrices semejantes). Cuando su determinante sea positivo pero su traza sea negativa, todos los valores propios serán negativos y la matriz será cóncava. Cuando el determinante sea negativo el método convergerá a un punto de silla. En estos casos basta con invertir el sentido de avance para alejarnos del máximo/punto de silla en vez de acercarnos. Cabe destacar que esta forma de proceder no es infalible: si nos encontramos en un punto intermedio entre un máximo y un punto de silla, el algoritmo podría, por ejemplo, empezar alejándose del máximo y, al caer en el entorno del punto de silla, empezar a alejarse de este y volver a caer en la región cóncava de la función.

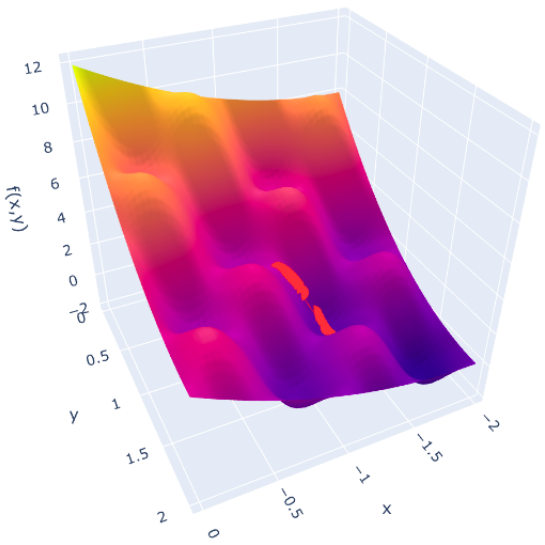
Para corregir el problema de la convergencia proponemos, por tanto, modificar la regla de avance original por la siguiente:

$$\Delta w = -\eta H_f(w)^{-1} \nabla f(w_o) \qquad \text{si } |H| > 0 \text{ y } tr(H) > 0$$

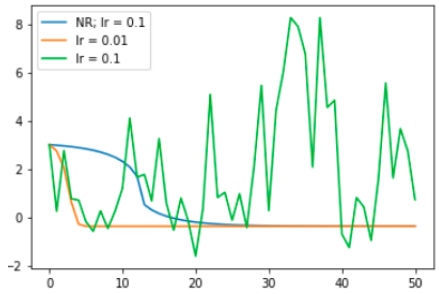
o bien,

$$\Delta w = -\eta H_f(w)^{-1} \nabla f(w_o) \qquad \text{en otro caso}$$

Repitiendo el experimento esta vez el proceso iterativo resulta así:



Vemos que, esta vez, sí parece que hay convergencia hacia un mínimo local. La solución encontrada es $(x, y) = (-1.26803, 1.28403)$ y $f(x, y) = -0.38092$. En la siguiente gráfica comparamos la evolución de f entre el método relajado de Newton con $\eta = 0.1$ y las pruebas con Gradiente Descendiente realizadas en el primer ejercicio:



Vemos que el método de Newton, para el mismo ratio de aprendizaje, es mucho más estable que el Gradiente Descendiente. Además, tiende a desacelerar su avance al acercarse al mínimo, haciendo uso de la información sobre la curvatura de la función proporcionada por la hessiana. El orden de convergencia de esta técnica, como ya hemos visto, es cuadrático (esto se aprecia claramente en la gráfica, si lo comparamos con el GD con $\eta = 0.1$). Como aspecto negativo, la velocidad de avance cerca de otros puntos críticos también se ve reducida, aunque no sean mínimos y busquemos alejarnos de ellos.

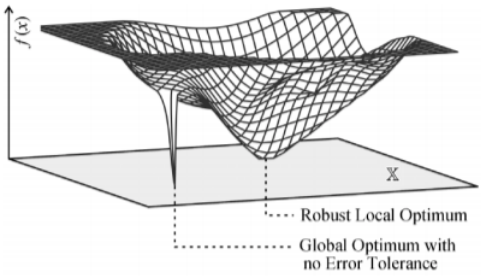
Usando nuestra versión modificada del método relajado de Newton, repetiremos el experimento realizado en el apartado sobre GD para distintos puntos iniciales, aunque conservando 0.1 como ratio de aprendizaje al habernos dado buenos resultados anteriormente. Los resultados pueden verse en la siguiente tabla:

Punto inicial	**Solución encontrada**	**f objetivo**
(-0.5,-0.5)	(-0.7916, -0.1302)	9.1257
(1.0,1.0)	(-0.7825, 2.7143)	0.5934
(2.1,-2.1)	(-20.4903, -18.6650)	1195.8747
(-3.0,3.0)	(-2.7320, 2.7160)	-0.3809
(-2.0,2.0)	(-2.0000, 2.0000)	-0.0000

Para el primer punto inicial, el 4º y el 5º, los resultados son los mismos que usando GD. Para el punto inicial $(1.0, 1.0)$, el resultado mejora usando el método de Newton, tal vez porque al usar un ratio de aprendizaje mayor nos hayamos saltado algún mínimo local. Sin embargo, el caso más preocupante es para el punto inicial $(2.1, -2.1)$, en el que observamos un valor de pérdida muy alto. Este resultado se debe, muy probablemente, a la inestabilidad intrínseca al uso de la inversa de la hessiana. En puntos donde la segunda derivada es muy baja (puntos donde f pasa de ser cóncava a ser convexa, por ejemplo), la inversa de la hessiana toma valores muy altos, por lo que la longitud del avance realizado por el método de Newton aumenta en la misma medida. Esto puede observarse superficialmente en el primera ejecución del método modificado, con $(-1.0, 1.0)$ como punto

inicial. En la gráfica en 3D observamos, en torno a la mitad del recorrido, que una de las iteraciones realiza un salto superior a los anteriores. Un fenómeno del mismo tipo podría haber sucedido en la tercera ejecución, aunque a mayor escala.

En general, el método de Newton parece ventajoso respecto al GD en casos en los que nos encontramos cerca del mínimo local y la función a optimizar es suficientemente suave. Dada la inestabilidad del método observada en las ejecuciones anteriores, este podría resultar contraproducente en funciones con cambios bruscos y frecuentes de la curvatura, o cerca de mínimos de tipo singularidad como vemos en la siguiente imagen comparativa obtenida de este artículo:



Conclusión: hemos estudiado empíricamente el comportamiento del método de Newton y lo hemos comparado con el Gradiente Descendiente para una función dada en forma analítica.

En general, el método de Newton parece un complemento adecuado para el GD cuando hemos logrado acercarnos lo suficiente a un mínimo como para aprovechar el conocimiento de la curvatura en su entorno, de forma que nos sea más fácil aproximarnos a este sin saltárnoslo.

Sin embargo, también han quedado patentes algunos inconvenientes del mismo que nos hacen preferir el GD en un contexto más general.

El primero es la tendencia del algoritmo original a aproximarse tanto a mínimos como a cualquier otro punto crítico, lo que nos ha obligado a aplicar cambios superficiales sobre el algoritmo original para facilitar la convergencia a mínimos. Además, incluso tras los cambios aplicados, la indiferencia del algoritmo ante distintos tipos de óptimos hace que su velocidad de avance se reduzca cerca de máximos locales y puntos de silla, incluso aunque el "paso" se de en la dirección contraria.

El segundo es la inestabilidad del procedimiento. Si nuestro punto de partida se encuentra en una región cóncava, es posible que al buscar el mínimo pasemos en busca del mínimo por zonas en las que la hessiana tome valores muy pequeños, lo que implica unos valores muy altos en su inversa, y una longitud de salto desproporcionada al acabar la iteración.

Referencias

- Plotly.
- Propiedades de las funciones convexas (*Wikipedia*).
- David H. Wolpert and William G. Macready, *No Free Lunch*.
- Yoshua Bengio, *Practical recommendations for gradient-based training of deep architectures*.
- Respuesta en StackExchange sobre casos de uso para la normalización.
- Tipos de puntos críticos según la hessiana.
- Weise T, Chiong R, Tang K. *Evolutionary optimization: Pitfalls and booby traps*.