2022/2023 1Q

Lista de Problemas 1

APA

Javier Béjar

Departament de Ciències de la Computació

Grau en Enginyeria Informàtica - UPC

UPC FIB Facultat d'Informàtica de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

DEPARTAMENT DE CIÉNCIES DE LA COMPUTACIÓ

FACULTAT D'INFORMÁTICA DE BARCELONA

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Primera edición, septiembre 2021

Esta edición, Septiembre 2022

Instrucciones:

Para la entrega de grupo debéis elegir un problema del capítulo de problemas de grupo.

Para la entrega individual debéis elegir un problema del capítulo de problemas individuales.

Cada miembro del grupo debe elegir un problema individual diferente.

Debéis hacer la entrega subiendo la solución al racó.

Evaluación:

La nota de esta entrega se calculará como 1/3 de la nota del problema de grupo más 2/3 de la nota del problema individual.

Al realizar el informe correspondiente a los problemas explicad los resultados y las respuestas a las preguntas de la manera que os parezca necesaria. Se valorará más que uséis gráficas u otros elementos para ser más ilustrativos.

La parte que no es de programación la podéis hacer a mano y escanearla a un archivo **PDF**. Comprobad que **sea legible**.

Para la parte de programación podéis entregar los resultados como un notebook (Colab/Jupyter). Alternativamente, podéis hacer un documento explicando los resultados como un PDF y un archivo python con el código

También, si queréis, podéis poner las respuestas a las preguntas en el notebook, este os permite insertar texto en markdown y en latex.

Aseguraos de que los notebooks mantienen la solución que habéis obtenido, no los entreguéis sin ejecutar.

Objetivos:

1. Saber calcular estimadores de máxima verosimilitud para distribuciones de probabilidad sencillas y conocer sus propiedades fundamentales.
2. Utilizar estimadores de máxima verosimilitud en problemas sencillos.
3. Utilizar diferentes funciones de pérdida
4. Programar algoritmos sencillos de optimización por descenso de gradiente.

CAPÍTULO 1

Problemas de Grupo

Sería conveniente que antes de resolver los problemas leáis las notas que tenéis en la web en el apartado del primer tema sobre inferencia estadística y estimación para refrescar vuestros conocimientos de la asignatura de estadística.

Para hacer la parte de implementación de estos ejercicios y algunos de los individuales deberéis leeros primero el capítulo 3 de este documento para aprender como usar optimización por descenso de gradiente usando `JAX`.

Si hacéis los experimentos en Colab usando GPU (o cualquier máquina configurada para usar la GPU instalada) y programando adecuadamente los algoritmos, la ejecución debería ser bastante rápida. En Colab ya os encontraréis `JAX` instalado. Aseguraos de que el entorno de ejecución que os asigne tiene GPU.

1. **Dos mejor que una**

   Consideremos un experimento en el que medimos una determinada variable aleatoria X, que sigue una distribución Gaussiana univariante ($X \sim \mathcal{N}(\mu, \sigma^2)$). Tomamos n medidas independientes de X y obtenemos una muestra aleatoria $\{x_1, \ldots, x_n\}$, donde cada x_i es una realización de X, para $i = 1, \ldots, n$. Resolved los siguientes apartados, ilustrando los resultados de la manera que os parezca más adecuada.

   a) Escribid la función de densidad de probabilidad para una x_i cualquiera y construid la función log-verosimilitud (negativa) de la muestra.

   b) Encontrad los estimadores de máxima verosimilitud $\hat{\mu}$ y $\hat{\sigma}^2$, a partir de la muestra.

   c) Demostrad que realmente son mínimos (y no extremos cualquiera).

   d) Implementad la función de log verosimilitud usando `JAX`. Definid muestras (independientes) de tamaños 50, 500 y 5000 para una distribución Gausiana con μ y σ de vuestra elección. Implementad un algoritmo de descenso de gradiente utilizando `JAX` para optimizar la log verosimilitud y estimar los parámetros de la distribución explorando la tasa de aprendizaje y el número máximo de iteraciones. Podéis emplear un valor de ϵ para acabar la optimización de

`1e-10` e inicializar los valores de los parámetros a un valor razonable que no este ni demasiado lejos, ni demasiado cerca de los valores reales. Comparad el resultado con el cálculo que dan los estimadores teóricos y comentad el resultado.

e) No todo se comporta según una distribución Gaussiana. A veces nuestros datos son generados por varios procesos diferentes que siguen distribuciones Gausianas con parámetros diferentes, básicamente tenemos una distribución conjunta con varias modalidades. Escribid la función de densidad de probabilidad para una distribución que sigue la suma de k Gaussianas (fijaos que ha de ser una distribución de probabilidad).

Supongamos que tenemos dos máquinas que fabrican dos tipos de tornillos de distinta longitud con cierta varianza también distinta. Todos los tornillos acaban mezclados y queremos calcular cuáles son los parámetros de la distribución de cada máquina. Sabemos que cada máquina ha fabricado el mismo número de tornillos. Generad dos muestras (independientes) de igual tamaño (2500) para cada distribución usando el generador de muestras normales de `JAX`. Mirad como se trata la generación de muestras en el capítulo 3 para que sean independientes. Implementad un algoritmo de descenso de gradiente para optimizar los parámetros de las dos distribuciones[1] con la muestra conjunta utilizando la función de log verosimilitud de la mezcla de dos gaussianas. Explorad diferentes valores para la tasa de aprendizaje y el número máximo de iteraciones.

f) Emplear la misma tasa de aprendizaje durante toda la optimización puede evitar que lleguemos realmente cerca del óptimo. Modificad el algoritmo que habéis implementado para que la tasa de aprendizaje vaya atenuándose con el número de iteraciones multiplicándola por un valor cercano a 1 (0,9999, 0.999, 0,99, ..., 0.9). Repetid la optimización de vuestro mejor resultado en el apartado anterior con diferentes atenuaciones. ¿Ha afectado al resultado? ¿Ha afectado al número de iteraciones antes de converger?

2. **El cuantizador que lo descuantize ...**

Una secuencia discreta está formada por símbolos (e.g. A, B, C, D, ...). Estamos interesados en la estimación de la probabilidad de cada símbolo que aparece en una secuencia. Trataremos cada posición como una variable aleatoria siguiendo una distribución categórica de k valores; las secuencias se forman por muestreo repetitivo (independiente) de esta distribución n veces. La distribución tiene k parámetros, que se denotan $p_1, \ldots, p_k$, tales que $p_i > 0$ y $\sum_{i=1}^{k} p_i = 1$ (asumiremos por ejemplo, $k = 4$). Resolved los siguientes apartados, ilustrando los resultados de la manera que os parezca más adecuada.

Pista: Consultad el libro de Bishop, página 75.

a) Tenemos una muestra $x_1, \ldots, x_n$, donde $x_i \in \{A, B, C, D\}$. Construid la función de log-verosimilitud negativa de la muestra.

b) Encontrad los estimadores de máxima verosimilitud para $p_i, i = 1, \ldots, 4$, a partir de la muestra.

c) Demostrad que realmente son máximos (y no extremos cualquiera).

d) El optimizar distribuciones con valores discretos es complicado, básicamente nos encontraremos con funciones que no son diferenciables en todos sus puntos o que tienen gradientes cero. Un truco usado en aprendizaje automático es transformar los valores a un rango continuo y aproximarla mediante una distribución continua que luego se puede transformar de manera sencilla en la distribución discreta. El proceso de pasar a valores continuos se llama decuantizacion. Esto además permite tratar variables categóricas ordenadas de una mejor manera, ya

[1] Pensad que `JAX` es suficientemente listo para calcular gradientes sobre un vector de parámetros.

que a una distribución categórica no le importa el orden de los símbolos. Esta decuantización se puede obtener simplemente añadiendo ruido uniforme a los valores discretos.

Esto es aplicable por ejemplo en problemas de imágenes, donde los valores de los píxeles son discretos y están ordenados. Tenéis en `scikit-learn` el conjunto de datos *digits*. Son imágenes de 8×8 con píxeles en 16 niveles de grises. Obtened los primeros 100 ejemplos de la clase 0, transformadlos en un vector y sumad al vector de valores un vector de muestras uniformes $u \sim U(-0{,}5, 0{,}5)$.

e) Los parámetros de la distribución categórica se pueden representar como un histograma. A medida que el número de categorías aumenta es poco práctico el estimar las probabilidades de cada categoría (pensad que en una imagen cada pixel está compuesto de 3 colores (RGB) con 256 niveles). Una aproximación más escalable es aprender una mezcla de d distribuciones continuas que aproximen el histograma (con $d \ll k$). Una posibilidad es usar la distribución logística:

$$f(x|\mu, \sigma) = \frac{\exp(\frac{-(x-\mu)}{\sigma})}{\sigma(1 + \exp(\frac{-(x-\mu)}{\sigma}))^2}$$

Define una función para la PDF y la log verosimilitud de esta distribución y la log verosimilitud de la mezcla de d funciones logísticas (fijaos que la PDF de la mezcla ha de ser una distribución de probabilidad).

f) Asumiendo que todas las distribuciones en la mezcla tienen el mismo peso, implementad un algoritmo de descenso de gradiente usando `JAX` y estimad los parámetros[1] μ y σ de las distribuciones logísticas con mezclas de 2, 4 y 8 distribuciones para la muestra que habéis obtenido de los datos de digits. Explorad diferentes valores para la tasa de aprendizaje y el número de iteraciones. Representad el histograma normalizado (frecuencias) que corresponde a los valores discretos y la PDF que corresponde a las mezclas de distribuciones logísticas. Comentad los resultados.

3. **Funciona hasta que al final se rompe**

La distribución de Poisson es una distribución discreta sobre conteos positivos. Se puede aplicar a sistemas con un gran número de posibles eventos, cada uno de los cuales es poco frecuente (por ejemplo, el número de errores diarios de entrada/saluda de un disco duro). Tiene como función de probabilidad:

$$p(x; \lambda) = e^{-\lambda} \frac{\lambda^x}{x!}, \quad x = 0, 1, 2, \ldots$$

donde x es el valor y $\lambda > 0$ el parámetro de la Poisson. Consideramos un experimento aleatorio en el que medimos una determinada variable aleatoria X, que sigue una distribución de Poisson ($X \sim Pois(\lambda)$). Tomamos n medidas independientes de X y obtenemos una muestra aleatoria simple $\{x_1, \ldots, x_n\}$, donde cada x_i es una realización de X, para $i = 1, \ldots, n$. Resolved los siguientes apartados, ilustrando los resultados de la manera que os parezca más adecuada.

a) Construid la función de log verosimilitud (negativa) de la muestra.

b) Encontrad el estimador de máxima verosimilitud para λ a partir de la muestra.

c) Demostrad que realmente es un mínimo (y no un extremo cualquiera).

d) Tenemos que estimar la log verosimilitud de la función Poisson para poder aplicar descenso de gradiente. Os habréis fijado que en esta distribución interviene la función *factorial*. Antes de que cunda el pánico, si no la conocéis ya, averiguad la relación de la función Gamma (Γ) y el factorial. Implementad la función de log verosimilitud usando `JAX`.

Generad muestras independientes de tamaños 100, 1000, 5000 para una distribución Poisson con un λ de vuestra elección usando el generador de muestras poisson de `JAX`. Mirad como se trata la generación de muestras en el capítulo 3 para que sean independientes. Implementad un algoritmo de descenso de gradiente usando `JAX` para optimizar la log verosimilitud y estimar el parámetro de la distribución explorando la tasa de aprendizaje y el número máximo de iteraciones. Podéis usar un valor de ϵ para acabar la optimización de `1e-10` e inicializar el valor del parámetro a un valor razonable que no este ni demasiado lejos, ni demasiado cerca del valor real. Comparad el valor estimado con el valor del estimador de máxima verosimilitud y con el valor que hayáis escogido. Comentad los resultados.

e) Muchas veces las probabilidades de los sucesos en un problema dependen de características de los ejemplos. Supongamos que queremos establecer la vida útil de un disco duro a partir de sus errores de entrada/salida. Claramente, la edad del disco duro influye en el número de errores de entrada salida. La regresión de Poisson asume una relación que es lineal entre los atributos de las muestras y el logaritmo del parámetro λ de manera que:

$$\lambda = \exp(w \cdot v)$$

donde ahora w son los parámetros de nuestra distribución y v es el vector de atributos asociado a los ejemplos. Asumiremos que la relación en nuestro problema es $(a + b \cdot t)$, de manera que al aumentar el tiempo (t) la tasa de error va creciendo. Generad muestras que sigan la relación $(2 + 0{,}05 \cdot t)$ tomando 50, 250 y 500 muestras a intervalos regulares en el intervalo $[0, 40]$. Estimad los parámetros a y b usando descenso de gradiente explorando la tasa de aprendizaje y el número máximo de iteraciones. Fijaos que ahora en la estimación tenéis dos valores de entrada t y x. Tendréis que ir con cuidado con la tasa de aprendizaje. Representad la evolución de la log verosimilitud y los parámetros.

f) En el apartado anterior habréis quizás observado entre otras cosas que el número de iteraciones necesarias para converger es relativamente largo. No siempre podemos asumir que la pendiente de la dirección de descenso del gradiente es igual en todas las dimensiones, a veces cada parámetro necesita su propia tasa de aprendizaje. Modificad el algoritmo de descenso de gradiente que habéis implementado en el apartado anterior y explorad diferentes tasas de aprendizaje para cada parámetro. Comentad los resultados.

4. **Lo nuclear ya es verde**

La distribución exponencial es una distribución continua definida sobre valores reales positivos. Es una distribución versátil donde la densidad de probabilidad cae de manera monótona. Tiene como función de probabilidad

$$p(x; \lambda) = \frac{1}{\lambda} e^{-\frac{x}{\lambda}}, \quad x > 0$$

donde x es el valor y $\lambda > 0$ el parámetro. Consideramos un experimento aleatorio en el que medimos una determinada variable aleatoria X, que sigue una distribución exponencial, cosa que escribimos $X \sim Exp(\lambda)$. Tomamos n medidas independientes de X y obtenemos una muestra aleatoria simple $\{x_1, \ldots, x_n\}$, donde cada x_i es una realización de X, para $i = 1, \ldots, n$. Resolved los siguientes apartados, ilustrando los resultados de la manera que os parezca más adecuada.

a) Construid la función de log verosimilitud (negativa) de una muestra.

b) Encontrad el estimador de máxima verosimilitud para λ a partir de la muestra

c) Demostrad que realmente es un mínimo (y no un extremo cualquiera).

d) Implementad la función de log verosimilitud usando `JAX`. Generad tres conjuntos de muestras independientes de tamaño 50, 500 y 2000 que sigan la distribución exponencial utilizando la función `jax.random.exponential` escogiendo un valor específico para el parámetro λ (el que queráis[2]). Mirad como se trata la generación de muestras en el capítulo 3 para que sean independientes.

e) Implementad un algoritmo de descenso de gradiente usando `JAX` para optimizar la log verosimilitud y estimad el parámetro de la distribución explorando la tasa de aprendizaje y el número máximo de iteraciones. Podéis utilizar un valor de ϵ para acabar la optimización de `1e-10` e inicializad el valor del parámetro a un valor razonable que no este ni demasiado lejos, ni demasiado cerca del valor real. Comparad el valor estimado con el valor del estimador de máxima verosimilitud y con el valor que hayáis escogido. Comentad los resultados.

f) Los procesos de desintegración radiactiva siguen una distribución exponencial. Los elementos radiactivos en muestras no suelen aparecer solos, es posible deducir cuáles son los elementos estimando la distribución conjunta de las desintegraciones en la muestra[3]. Supondremos que tenemos una muestra en la que sabemos que hay exactamente dos elementos radiactivos que están en la misma proporción (50 %/50 %). Esto definiría una distribución conjunta con esta expresión:

$$p(x;\lambda_1,\lambda_2) = 0{,}5 \cdot \frac{1}{\lambda_1} e^{-\frac{x}{\lambda_1}} + 0{,}5 \cdot \frac{1}{\lambda_2} e^{-\frac{x}{\lambda_2}}$$

Elegid dos valores para los parámetros λ_1 y λ_2 que sean diferentes. Generad una muestra que combine dos muestras independientes del mismo tamaño (2500) generadas con cada distribución exponencial. Estimad los dos parámetros usando descenso de gradiente estudiando el efecto de la tasa de aprendizaje. Fijaos que es una función de dos parámetros por lo que tenéis que calcular derivadas parciales respecto a cada parámetro. La función `grad` tiene un parámetro `argnums` donde se puede indicar el número del parámetro sobre el que calcular el gradiente. Representad como evolucionan los parámetros y la función de log verosimilitud con las iteraciones. Comentad los resultados.

[2] Fijaos que la función de `JAX` que genera muestras distribuidas según la exponencial solo genera muestras para $\lambda = 1$, pensad como generar muestras para λ arbitraria. Como pista, pensad que las muestras para $\lambda = 1$ se generan empleando la función inversa de la distribución con muestras generadas a partir de la distribución uniforme.

[3] Asumiremos que las emisiones de la desintegración de los dos elementos son las mismas.

CAPÍTULO 2

Problemas Individuales

En los problemas en los que hay que usar `JAX` es mejor resolverlo usando Colab dado que podréis usar una GPU y los experimentos irán más deprisa. Para el resto de problemas no debería haber gran diferencia.
En Colab ya os encontraréis `JAX` instalado. Aseguraos de que el entorno de ejecución que os asigne tiene GPU.

1. **Seamos sensibles, pero solo un épsilon**

   El error cuadrático es la función de pérdida habitual para medir la calidad del ajuste de una función a unos datos, pero no siempre es la mejor para encontrar los parámetros adecuados. Hay problemas en los que los errores pequeños no son importantes y preferimos ignorarlos, prefiriendo que el ajuste se guíe por ejemplos con mayor error. Una función de pérdida que permite ignorar errores es la ϵ-insensitive loss definida como:

   $$\ell(y_n, f(x_n), \epsilon) = \text{máx}(0, |y_n - f(x_n)| - \epsilon)$$

   El objetivo del problema es optimizar una regresión de una variable definida como:

   $$f(x) = w_0 + w_1 x$$

   Para optimizar los parámetros usaremos descenso de gradiente utilizando `JAX` igual que habeis hecho en los problemas de grupo.

   a) Representa la función de pérdida ϵ-insensitive para diferentes valores de ϵ y el error cuadrático en el rango $[-3, 3]$. ¿que ventaja te parece que tiene esta función al ignorar los errores menores que un cierto valor?

   b) Define un problema de regresión eligiendo dos valores para los parámetros w_0 y w_1. Genera tres conjuntos de datos de 100 ejemplos usando la distribución uniforme en el rango $[0, 20]$, han de ser tres muestras independientes (mira como se generan muestras independientes con

los generadores de `JAX` en el capítulo 3). Los dos primeros nos servirán de conjunto de entrenamiento y el último como conjunto de test. Calcula la salida para cada ejemplo usando la función:

$$f(x) = w_0 + w_1 x + \gamma$$

Donde γ es ruido gausiano que generaremos de dos maneras diferentes, uno será $\gamma \sim \mathcal{N}(0, 1)$ y el otro $\gamma \sim \mathcal{N}(0, 5)$. Añade el primer ruido al calcular $f(x)$ para primer conjunto de datos y el segundo a los otros dos. Junta los dos primeros conjuntos de datos para tener el conjunto de entrenamiento.

c) Para encontrar los valores de los parámetros implementa un algoritmo de descenso de gradiente con `JAX` usando la función de pérdida ϵ-insensitive. En este caso el usar una tasa de aprendizaje fija puede causar problemas al optimizar. La calcularemos en función de las iteraciones t de la siguiente manera:

$$lr(t) = \frac{lr_0}{1 + lr_0 \cdot \alpha \cdot t}$$

Donde lr_0 es una tasa de aprendizaje inicial y α es un hiperparámetro que da un control adicional sobre como desciende la tasa de aprendizaje.

Encuentra el mejor conjunto de parámetros explorando los valores $[10, 1, 0{,}1, 0{,}01, 0{,}001]$ para ϵ, lr_0 y α. Selecciona el mejor conjunto usando el error cuadrático sobre el conjunto de test como criterio. Escoge un número máximo de iteraciones y un valor para decidir el final de la optimización que te parezcan adecuados.

d) Repite el experimento del apartado anterior ajustando los parámetros de la regresión usando el error cuadrático en lugar de la pérdida ϵ-insensitive y seleccionando los mejores también con el error cuadrático en el test. Compara los resultados con los obtenidos en el apartado anterior.

e) Nadie dice que en la optimización no podamos usar una función de pérdida para decidir el cambio en los parámetros y usar otra para decidir cuando acabar la optimización. Repite los experimentos usando el gradiente de la pérdida ϵ-insensitive para ajustar los parámetros y el error cuadrático para acabar la optimización. Selecciona los mejores parámetros de igual manera que en los apartados anteriores. Compara con los resultados anteriores. ¿Por qué no sería buena idea el seleccionar los mejores parámetros usando la pérdida ϵ-insensitive en el conjunto de test?

2. **Hay que tener confianza, intervalo de confianza**

Al ajustar una función solo tenemos una predicción puntual para las predicciones que hacemos. En algunos modelos sencillos es posible definir cuál es la distribución de los estimadores, pero con una función arbitraria puede ser imposible calcularlo analíticamente. La función de pérdida *cuantil* o *pinball loss* permite ajustar una función a diferentes cuantiles de los datos para poder obtener funciones que correspondan a los cuantiles de la distribución de los datos. A partir de ellas podemos obtener un valor máximo y mínimo de predicción que corresponda a un rango de cuantiles específicos. Esta función se define como:

$$\ell(y_n, f(x_n), q) = q \cdot \text{máx}(0, y_n - f(x_n)) + (1 - q) \cdot \text{máx}(0, f(x_n) - y_n)$$

Donde q es el cuantil que queremos calcular.

a) Comprueba que esta función corresponde con el error medio absoluto (MAE) cuando $q = 0{,}5$.

b) Define un problema de regresión de una variable eligiendo dos valores para los parámetros w_0 y w_1. Genera un conjunto de datos de 100 ejemplos usando la distribución uniforme en el rango $[0, 20]$. Calcula la salida para cada ejemplo usando la función:

$$f(x) = w_0 + w_1 x + \epsilon$$

Donde ϵ es ruido gausiano que generaremos como $\epsilon \sim \mathcal{N}(0, 2)$

c) Para encontrar los valores de los parámetros implementa un algoritmo de descenso de gradiente con `JAX` usando la función de pérdida cuantil. En este caso el usar una tasa de aprendizaje fija puede hacer lenta la optimización. La calcularemos en función de las iteraciones t de la siguiente manera:

$$lr(t) = \frac{lr_0}{(1+t)^p}$$

Donde lr_0 es una tasa de aprendizaje inicial y p es un hiperparámetro que da un control adicional sobre como desciende la tasa de aprendizaje. Encuentra el mejor conjunto de parámetros para $q \in [0{,}9, 0{,}5, 0{,}1]$ explorando valores para lr_0 y p. Comenta lo que observes en el comportamiento del error y los parámetros durante la optimización. Escoge un número de iteraciones y un valor para decidir el final de la optimización que te parezcan adecuados.

d) Dado que el ruido es homocedástico (siempre la misma varianza) no debería serte difícil saber cual debería ser la distancia (en vertical) entre la regresión con $q = 0{,}9$ y $q = 0{,}1$. Calcula la distancia entre las dos rectas de regresión y comprueba que esa distancia es aproximadamente la que debería.

e) La gracia de esta función de pérdida es calcular intervalos de confianza cuando tenemos un modelo arbitrario o cuando no es cierto que el comportamiento del ruido siga una distribución normal homocedástica. Genera una nueva muestra de datos donde la varianza del ruido sea una función lineal de la variable (básicamente que aumente o disminuya dependiendo de su valor). Ajusta los mismos cuantiles que en apartado c) y comprueba que la distancia entre los valores que corresponden a los cuantiles extremos corresponden a la función que has usado para generar la varianza del ruido.

3. **¿Kullback que?**

Cuando trabajamos con modelos que representan una distribución de probabilidad nuestro objetivo es hacer que la distribución de los datos se acerque lo más posible a las probabilidades que nos da el modelo sobre esos datos. Existen muchas maneras de calcular esa diferencia, una común es usar funciones de divergencia, entre ellas la divergencia de Kullback-Leibler es la más usada. Dadas dos distribuciones de probabilidad P y Q se define asumiendo que sean distribuciones discretas como:

$$KL(P|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

En el caso de distribuciones continuas, simplemente substituimos el sumatorio por una integral.

a) Siendo X una muestra de datos $x_1, \ldots, x_n$ de valores discretos, donde podemos estimar su distribución P a partir de su frecuencia y Q es una distribución de probabilidad sobre el mismo rango de valores discretos. Demuestra que optimizar $KL(P|Q)$ es equivalente a optimizar la log verosimilitud negativa de Q sobre los datos.

b) Todo modelo de clasificación es una distribución de probabilidad sobre un conjunto de valores discretos, por lo que podemos ajustar un modelo probabilístico para clasificación haciendo que las probabilidades que obtenga para una muestra se ajusten a las de los datos. Usa la función

`make_classification` de scikit-learn para crear un conjunto de datos de clasificación de dos dimensiones y 100 ejemplos. Tendrás dar un valor 0 al parámetro `n_redundant` y un valor 1 al parámetro `n_clusters_per_class`. Da un valor también al parámetro `random_state` para que los experimentos sean reproducibles. El problema que generará será de clasificación binaria.

c) Podemos crear un modelo probabilístico con una función linear $f(w, x) = w \cdot x$. Para obtener probabilidades simplemente tenemos que aplicar sobre el resultado una función que de un valor entre 0 y 1. Por ejemplo la función sigmoide σ:

$$\sigma(x) = \frac{1}{1 + e^x}$$

A partir de la divergencia de Kullback-Leibler simplificando para problemas binarios podemos llegar a la función de pérdida de entropía cruzada binaria (binary cross entropy):

$$BCE(p(x), y) = y * \log(p(x)) + (1 - y)log(1 - p(x))$$

Donde $p(x)$ es la probabilidad que le asigna el modelo a un ejemplo, e y es la etiqueta que le corresponde a los datos. Implementa un algoritmo de descenso de gradiente usando `JAX` con la función de entropía cruzada binaria. Explora diferentes tasas de aprendizaje. Comenta lo que observes en el comportamiento del error y los parámetros durante la optimización. Escoge un número de iteraciones y un valor para decidir el final de la optimización que te parezcan adecuados.

d) Genera un conjunto de datos con la función `make_circles` de scikit-learn usando el valor 0,1 para el parámetro `noise`. Optimiza el modelo para varios parámetros iniciales diferentes del modelo. Cuenta que esta que sucediendo e intenta explicar el porqué.

e) La función de entropía cruzada parece una función extraña para optimizar cuando lo que nos interesa es un modelo que tenga el mínimo número de ejemplos mal clasificados. En este caso se correspondería a la función de pérdida 0/1, que en el caso de probabilidades asignaría una pérdida de 0 a valores menores que 0.5 y 1 en caso contrario ¿Porqué no es una buena idea optimizar directamente esta función? Representa las dos funciones.

4. **El error cuadrático no es todo lo que hay**

El error cuadrático es una de las funciones de pérdida habituales que se optimiza cuando se aprenden modelos basados en funciones, pero no siempre tiene buenos resultados dependiendo de las características de los datos. Uno de sus principales problemas son los valores atípicos, ejemplos que tienen valores de salida fuera del rango normal.

El propósito de este problema es experimentar con la función de pérdida *log cosh* definida como:

$$\ell(y_n, f(x_n)) = \log cosh(y_n - f(x_n)) = \log\left(\frac{e^{(y_n - f(x_n))} + e^{-(y_n - f(x_n))}}{2}\right)$$

El objetivo es optimizar una regresión de una variable definida como:

$$f(x) = w_0 + w_1 x$$

Vamos a usar la función que proporciona la biblioteca `scipy` para la minimización de funciones no lineales (`scipy.optimize.minimize`) que optimiza una función que recibe un vector de parámetros a optimizar, en este caso queremos minimizar la pérdida que será función de dos parámetros (w_0,w_1). El método de optimización a utilizar es `BFGS`[1].

[1] https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html#broyden-fletcher-goldfarb-shanno-alg

a) Representa la función de pérdida log cosh y el error cuadrático en el rango $[-3, 3]$ y explica por qué crees que log cosh es más tolerante a los valores atípicos. El error cuadrático se define como:

$$\ell(y_n, f(x_n)) = \frac{1}{2}(y_n - f(x_n))^2$$

b) Define un problema de regresión eligiendo dos valores para los parámetros w_0 y w_1. Genera dos conjuntos de datos de 100 ejemplos usando la distribución normal en el rango $[-5, 5]$ y calcula la salida para cada ejemplo usando la función:

$$f(x) = w_0 + w_1 x + \epsilon$$

donde $\epsilon \sim \mathcal{N}(0, 2)$. Para el primer conjunto de datos vamos a generar ejemplos atípicos añadiendo al 10 % de los ejemplos un ruido normal adicional $\epsilon' \sim \mathcal{N}(10, 2)$.

c) Encuentra la derivada de las dos funciones de pérdida (tienes la derivada del error cuadrático en las diapositivas de teoría).

d) Para encontrar el mínimo de las funciones de pérdida, debes definir dos funciones Python, una para la pérdida y otra para su derivada (este será el parámetro `jac` de la función de minimización). Las funciones deben definirse para una muestra y las predicciones correctas (que se reciben como parámetros adicionales a la función para minimizar), suponiendo que las variables `X` e `y` son los datos y la salida deseada, la función para el error cuadrático debería verse así:

```
def se(w, X, y):
        return 0.5 * np.sum(y - (w[0] + (w[1] * X))**2)
```

Recuerda que tienes funciones de dos parámetros, por lo que las derivadas devuelven dos valores.

e) Calcula los parámetros de la muestra minimizando las dos funciones de pérdida usando el mismo valor inicial para los parámetros de ambos y representa los resultados. ¿Qué conjunto de parámetros parece mejor? ¿Por qué? Calcula la pérdida cuadrática para ambas soluciones en el segundo conjunto de datos y comenta los resultados.

5. **La distribución normal no se ajusta a todos los datos**

Estamos acostumbrados a suponer que los datos siguen la distribución normal, pero hay dominios donde esto es obviamente incorrecto. Por ejemplo, problemas donde las variables tienen un máximo y un valor mínimo impuesto por la naturaleza de los datos y donde las probabilidades están sesgadas hacia uno de los extremos. En ese caso podemos ajustar una distribución más adecuada. Por ejemplo, la distribución **Gumbel** se usa específicamente para ese tipo de datos. La función de densidad de probabilidad de la distribución de Gumbel se define como:

$$p(x : \mu, \beta) = \frac{1}{\beta} \exp\left(-\frac{(x-\mu)}{\beta}\right) \exp\left(-\exp\left(-\frac{(x-\mu)}{\beta}\right)\right)$$

a) Define la función de distribución de probabilidad como una función Python y representa la distribución para diferentes parámetros.

b) Escribe el logaritmo de la verosimilitud negativa de una muestra y encuentra las derivadas parciales con respecto a los parámetros.

c) Las derivadas parciales dan como resultado un sistema de ecuaciones no lineales que deben resolverse numéricamente. Por el mismo precio podemos ajustar directamente los parámetros de la distribución mediante optimización numérica. Vamos a usar la función que proporciona la biblioteca `scipy` para la minimización de funciones no lineales

(`scipy.optimize.minimize`) que optimiza una función que recibe un vector de parámetros, en este caso queremos minimizar el negativo del logaritmo de la probabilidad que será una función de los dos parámetros (μ,β). El método de optimización que se debe usar es `Nelder-Mead`[2]. Tendrás que definir una función de python para el negativo del logaritmo de la probabilidad que recibirá un vector con los parámetros y la muestra de datos.

Aplicaremos esta distribución al conjunto de datos `California Housing` que se puede descargar usando la función `fetch_california_housing` de la biblioteca `scikit-learn`. Si representas las variables, verás que hay varias variables que se comportan como la distribución de Gumbel. Elige dos de estas variables y utiliza la función de minimización para ajustar la verosimilitud logarítmica negativa de la distribución de Gumbel a los datos de estas variables.

d) Representa la distribución de Gumbel para los parámetros ajustados contra el histograma normalizado de los datos. ¿Los parámetros se ajustan bien a los datos?

e) Ahora calcula la media y la varianza de las variables asumiendo que son gausianas y calcula el logaritmo de la probabilidad que las distribuciones gausiana y Gumbel asignan a una muestra de los datos (puedes usar la función `scipy.stats.norm.pdf` para la gausiana). ¿Hay una gran diferencia en las probabilidades?

[2] https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html#nelder-mead-simplex-algoritmo-método

CAPÍTULO 3

Diferenciación automática con JAX

3.1. Optimización de funciones y descenso de gradiente

El elemento clave del entrenamiento de modelos de aprendizaje automático es el ajuste de sus parámetros. Un modelo puede estar formado por funciones arbitrarias sobre las que, durante el proceso de aprendizaje, calculamos cuál es la función de pérdida que corresponde a un conjunto de parámetros y que valores permiten minimizarla.

En la mayoría de los casos la función de perdida aplicada al modelo de aprendizaje corresponde a una función diferenciable (no tiene por que serlo en todo su ámbito) y podríamos ajustar sus parámetros simplemente calculando la expresión de su derivada e igualándola a cero para calcular su expresión analítica. Desafortunadamente, no siempre es posible hacerlo, por lo que debemos obtener su solución aproximada usando métodos numéricos.

Es probable que conozcáis ya algún método numérico de optimización de funciones por ejemplo el método de Newton u otros métodos similares. Muchos de ellos son computacionalmente caros, especialmente si se trata de funciones con muchos parámetros, ya que suelen involucrar el cálculo del Hessiano de la función, lo que hace que escalen cuadráticamente con el número de parámetros. Para funciones muy complejas en las que hay muchos óptimos posibles, con muchos parámetros el coste de la optimización será importante.

La optimización por *descenso de gradiente* (*gradient descent*) aprovecha que podemos calcular la derivada puntual de la función y averiguar en que dirección desciende. Siguiendo esa dirección podemos intentar acercarnos al conjunto de parámetros que la hace cero.

Si recordáis los algoritmos de búsqueda local que visteis en la asignatura de Inteligencia Artificial (Hill Climbing, Simulated Annealing), estamos siguiendo el mismo principio, con la ventaja de que por lo general ahora estaremos trabajando con funciones continuas, lo que nos permite obtener directamente la dirección de máxima variación gracias a la diferenciacion.

A diferencia de lo que hacíais en IA, ahora el espacio es continuo en lugar de discreto (no hay operadores de búsqueda), por lo que tenemos que decidir también cuanto nos movemos en la dirección de máxima variación. Esto corresponde a un parámetro a decidir en la optimización que puede ser un valor fijo o ser adaptativo. En aprendizaje automático este parámetro se suele denominar *tasa de aprendizaje*

(*learning rate*).

En los algoritmos de aprendizaje el espacio de búsqueda corresponde a los parámetros, la función a optimizar será la función de pérdida definida sobre las evaluaciones del modelo en el conjunto de ejemplos. El objetivo es encontrar los parámetros para el modelo que minimizan esas evaluaciones de la función de pérdida. Es lo que hemos visto que buscamos en la minimización del riesgo empírico o en la minimización del logaritmo de la función de verosimilitud.

El algoritmo de descenso de gradiente simplemente va moviéndose en el espacio de parámetros en la dirección del gradiente dando pasos sucesivos con una longitud determinada por la tasa de aprendizaje. Podemos definirlo algorítmicamente de la siguiente manera, dada una función `f` con parámetros `p` y datos de entrada `x`:

```
repetir
    para cada parametro pi en p;
        grad[pi] = diferenciacion(f(pi))(x)
        mgrad[pi] = media(grad[pi])
        pi = pi - (tasa_aprendizaje * mgrad[pi])
hasta que f(pi_anteriores,x) - f(pi,x)) < epsilon
```

En nuestro caso `f` será la función de pérdida. Su gradiente nos dice en que dirección (en media) el cambio de los parámetros reduce su valor en el conjunto de datos. Si reajustamos los parámetros en esa dirección la nueva función estará más cerca de un óptimo. Fijaos que el gradiente se calcula en cada dato del conjunto de entrenamiento, su media nos da la esperanza matemática de la dirección en la que hemos de movernos. Fijaos también que así estamos aprendiendo los parámetros del modelo que se ajusta lo mejor posible a los datos observados.

3.2. JAX y diferenciación automática

Obviamente, el tener que calcular la derivada de una función arbitraria fácilmente es la clave de este método (y otros métodos similares). Podemos calcular la derivada analíticamente. Existen programas capaces de encontrar la expresión de la derivada de una función, pero dada una función arbitraria su expresión puede ser demasiado compleja. La podemos calcular numéricamente, aprovechando que la derivada de una función en un punto se puede calcular como $f(x) - f(x - \epsilon)$ para valores pequeños de ϵ, pero puede ser costoso. Una tercera vía es usar lo que se denomina diferenciación automática. Una función arbitraria está formada por la composición de funciones primitivas de las cuales conocemos su derivada. Podemos considerar que una función es básicamente un grafo de computación que nos indica como se realiza esa composición. Esto nos lleva a la posibilidad de obtener algorítmicamente la derivada de cualquier función a partir de ese grafo de computación.

No explicaremos aquí las interioridades de como funcionan los algoritmos de diferenciación automática, podéis haceros una idea consultando el enlace de wikipedia. En esencia, se basa en la aplicación recursiva de las reglas de derivación sobre el grafo que define la función en un paso hacia adelante para calcular las derivadas individuales y un paso hacia atrás acumulando los resultados calculados en el paso hacia adelante.

`JAX` es una librería para computación diferenciable desarrollada por Google DeepMind que permite entre otras cosas definir algoritmos de aprendizaje y usar optimización para ajustar sus parámetros. Veremos con un ejemplo los elementos básicos que permiten usarlo para optimizar funciones.

Una parte de `JAX` replica (casi) la librería `numpy`, lo que hace más fácil su uso para definir funciones numéricas. La principal ventaja es que (aparte de la diferenciación automática) permite el uso de acele-

ración por hardware (GPU, TPU, ejecución distribuida) de manera transparente, lo que permite acelerar los cálculos si se dispone de esa posibilidad.

Importaremos primero la versión de `numpy` de `JAX` y las funciones para calcular la derivada de una función (`grad`), la función de vectorización (`vmap`) que vectoriza la aplicación de una funcion, el compilador just in time (`jit`) y el generador de números aleatorios (`random`).

```
import jax.numpy as jnp
from jax import grad, vmap, jit, random
```

Definimos una función parametrizada (`myfunction`) que queremos usar para ajustar a un conjunto de datos, en este caso $f(x) = a \cdot \cos(x) + b \cdot \sin(3 \cdot x)$. La función python que definimos tiene dos parámetros, el vector de parámetros que ajustaremos (`p` = [a,b]) y los datos (`x`). Definimos a su vez dos funciones de pérdida, el error absoluto medio (MAE) y el error cuadrático medio (MSE). `JAX` permite emplear un compilador *just in time* (jit) para acelerar la ejecución, podemos activarlo utilizando el decorador `@jax.jit`.

```
@jax.jit
def myfunction(p,x):
    return jnp.sum(p * jnp.array([jnp.cos(x),jnp.sin(3*x)]).T)

@jax.jit
def lossMAE(p,x,y):
    return jnp.abs(myfunction(p,x) - y)

@jax.jit
def lossMSE(p,x,y):
    return (myfunction(p,x) - y)**2
```

Generamos un conjunto de datos evaluando la función en el rango $[-2, 2]$ como una muestra con ruido gausiano ($\mathcal{N}(0, 0{,}3)$). Calculamos los valores aplicando la función de vectorización sobre la función que queremos optimizar con unos parámetros específicos $[1, 0{,}7]$. Calculamos la función sobre el conjunto de datos usando `vmap` que vectoriza el cálculo, indicándole que la vectorización se ha de hacer sobre el segundo parámetro con el patrón que le indicamos en el parámetro `in_axes`.

El valor que se pasa en `in_axes` indica como se ha de hacer la vectorización para cada uno de los parámetros de entrada. Será una tupla con tantos elementos como parámetros tenga la entrada de la función, donde podemos indicar `None` en las posiciones de los parámetros que son escalares o que se han de usar tal cual para todas las aplicaciones de la función o un número que indica el eje sobre el que se ha de aplicar la función sobre la matriz que se pasa en ese parámetro.

Por ejemplo, para una función f con dos parámetros, donde el primero es un escalar y el segundo es una matriz sobre la que la vectorización se ha de aplicar en el primer eje indicaremos `in_axes=(None, 0)`.

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(10,10))
key = random.PRNGKey(0)
x = jnp.linspace(-2,2)
data = vmap(myfunction, in_axes=(None,0))(jnp.array([1,0.7]), x)
y = data +  (random.normal(key,(50,)) *0.3)
```

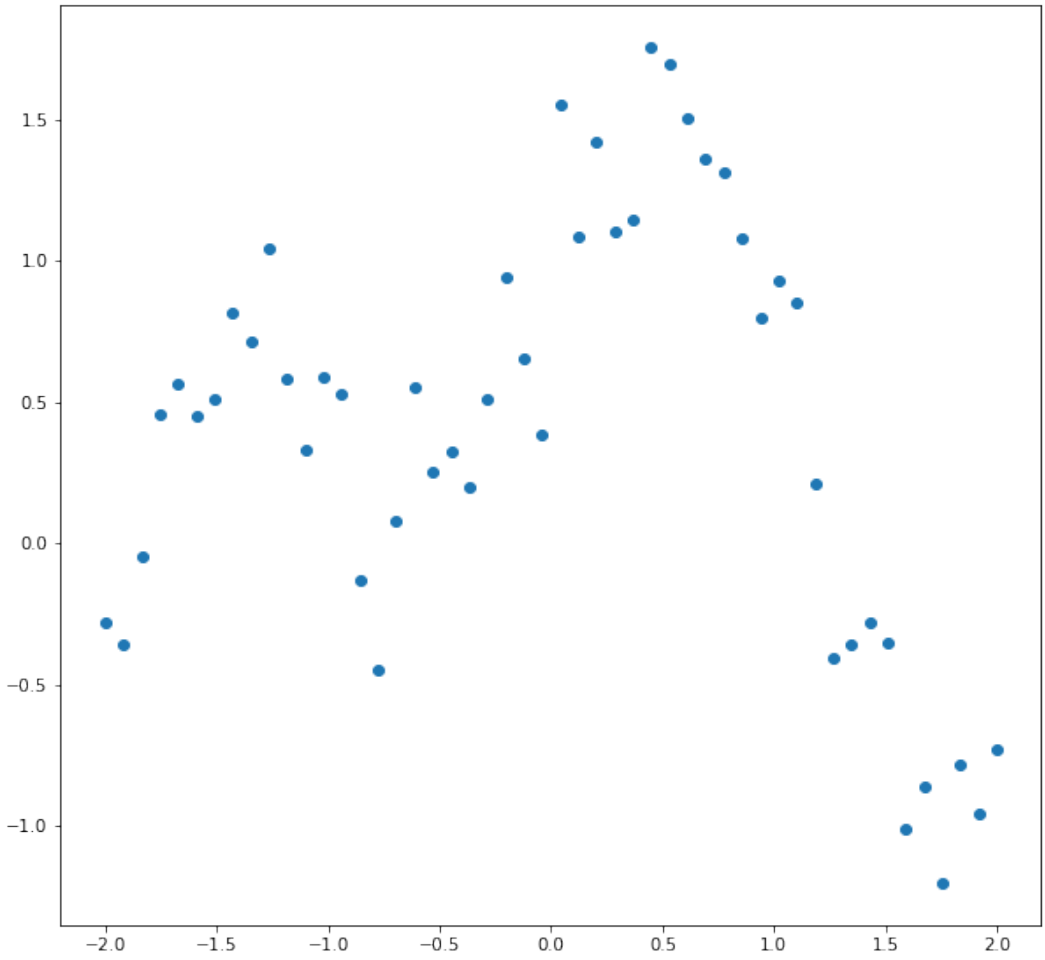
```
plt.scatter(jnp.linspace(-2,2),y)
```

Como nota la margen, la gestión de la generación de números aleatorios en JAX es diferente que en numpy, JAX está pensada con una filosofía funcional, todo es inmutable y sin efectos laterales en la ejecución, por lo que el generador de números aleatorios no tiene estado. Si utilizamos la clave obtenida del generador con una semilla siempre obtendremos los mismos números aleatorios. Para obtener secuencias diferentes a partir de la misma semilla, tenemos que *partir* la secuencia en diferentes claves, eso lo podemos hacer mediante la función `split`. Por ejemplo, si queremos tener dos secuencias independientes de números aleatorios a partir de la misma semilla haremos:

```
key1, key2 = random.split(jax.random.PRNGKey(0),2)
```

Esto nos dará dos claves que permiten generar dos secuencias con la misma semilla. Esto hace que tengamos que controlar explícitamente como generamos los números aleatorios de nuestros experimentos, pero permite una mejor reproducibilidad.

Estos son los ejemplos con los que haremos la optimización.



Aplicaremos el esquema correspondiente de descenso de gradiente. Debemos definir el número máximo de iteraciones que haremos en la búsqueda (1000) y la diferencia de la función de perdida entre dos iteraciones sucesivas para poder acabar cuando converjamos (`epsilon`). Iniciamos desde una solución arbitraria (`[0.5,1]`). Fijamos la tasa de aprendizaje a un número no demasiado pequeño para que no tarde demasiado, ni demasiado grande para que la búsqueda diverja. Este lo ajustaremos experimentando

un rango de valores hasta encontrar uno que funcione suficientemente bien, es un hiperparámetro del algoritmo. La función `grad` nos retorna la derivada (calculada por diferenciación automática) como una función, que aplicamos a los datos para obtener el gradiente en todos los puntos del conjunto de datos. Para obtener una función que se ejecute más rápido usamos la función `jit` que nos compilará la derivada de la función. Empezaremos optimizando el error cuadrático:

```
epsilon = 1e-9
lr = 0.01
paramMSE=jnp.array([0.5,1])
grad_MSE = jit(grad(lossMSE))
ploss = vmap(lossMSE,in_axes=(None,0,0))(paramMSE,x,y).mean(0)
for i in range(1000):
     part = vmap(grad_MSE,in_axes=(None,0,0))(paramMSE,x,y)
     paramMSE -= (lr * part.mean(0))
     loss = vmap(lossMSE,in_axes=(None,0,0))(paramMSE,x,y).mean(0)
     if jnp.abs(ploss - loss) < epsilon:
         print(i)
         break
     ploss = loss
print(paramMSE, loss)
```

```
745
```

```
[1.0165633 0.6869234] 0.06674306
```

Obtenemos una solución que converge en la iteración 753 que está suficientemente cerca del valor real. Fijaos que al haber introducido ruido en los valores de la función no obtendremos los valores exactos, sino el valor de los parámetros que reduzca más el error.

Ahora optimizaremos los parámetros usando el error absoluto. Partiremos de la misma solución inicial con la misma tasa de aprendizaje.

```
epsilon = 1e-9
lr = 0.01
paramMAE=jnp.array([0.5,1])
ploss = vmap(lossMAE,in_axes=(None,0,0))(paramMAE,x,y).mean(0)
for i in range(1000):
    part = vmap(grad(lossMAE),in_axes=(None,0,0))(paramMAE,x,y)
    paramMAE -= (lr * part.mean(0))
    loss = vmap(lossMAE,in_axes=(None,0,0))(paramMAE,x,y).mean(0)
    if jnp.abs(ploss -loss) < epsilon:
        print(i)
       break
    ploss = loss
print(paramMAE, loss)
```

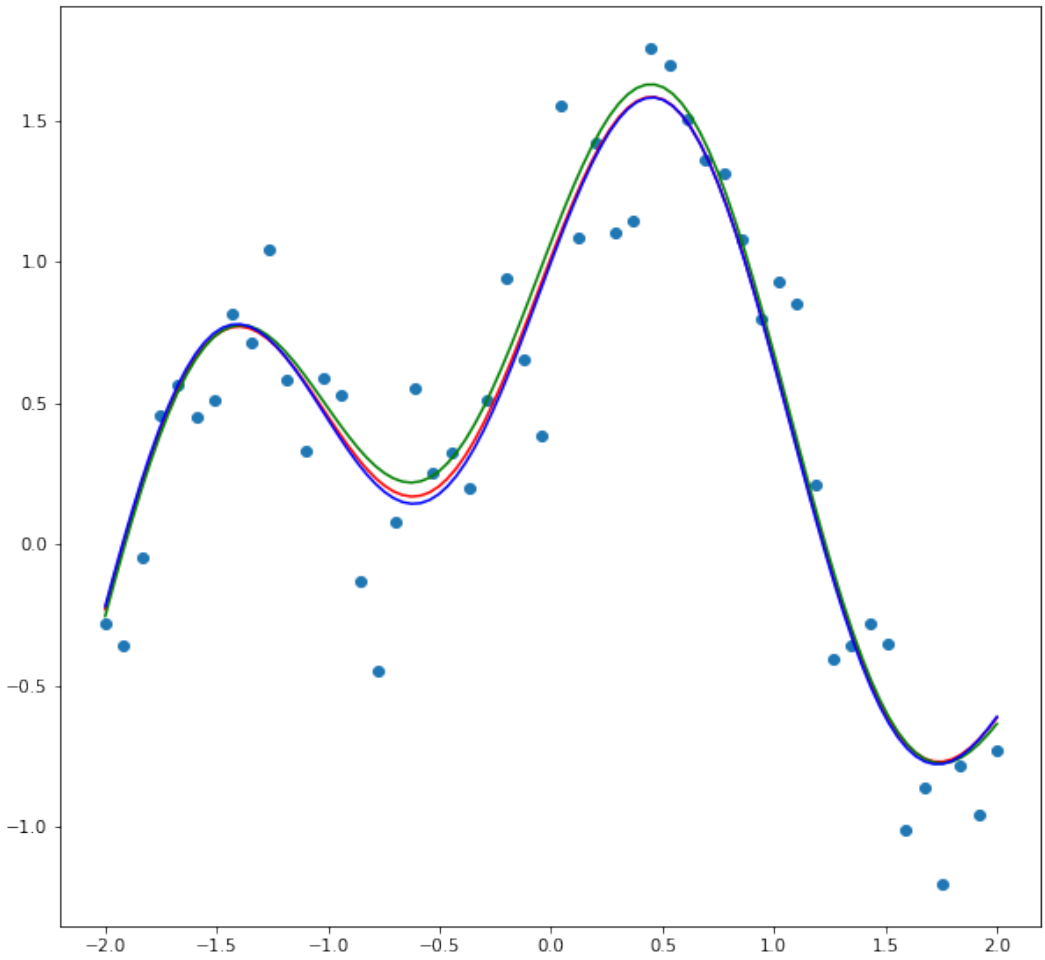
```
510
```

```
[1.0728726 0.6831946] 0.19826938
```

Los parámetros no tienen por qué salir iguales dado que estamos pesando los errores de diferente manera, el error cuadrático intentará reducir mucho más los errores más grandes, ya que están magnificados con el cuadrado.

Podemos representar las funciones que corresponden a los parámetros de las dos optimizaciones.

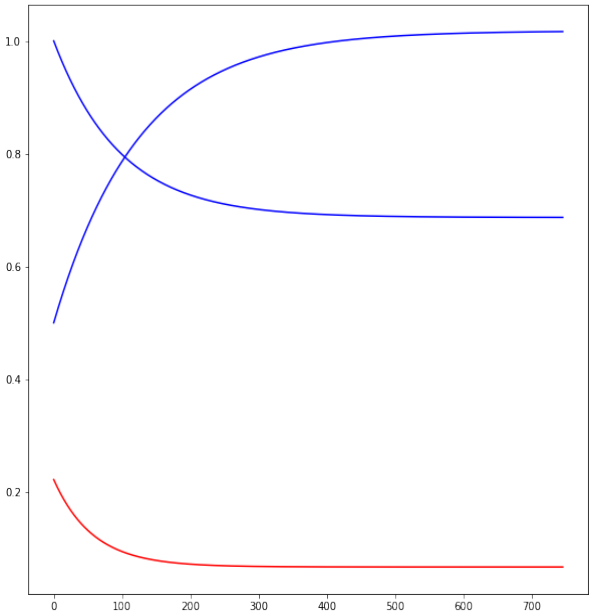
```
fig = plt.figure(figsize=(10,10))
xlim = jnp.linspace(-2,2,100)
dataMSE = vmap(myfunction, in_axes=(None,0))(paramMSE, xlim)
dataMAE = vmap(myfunction, in_axes=(None,0))(paramMAE, xlim)
real = vmap(myfunction, in_axes=(None,0))(jnp.array([1,0.7]), xlim)
plt.plot(xlim,dataMSE, 'r')
plt.plot(xlim,dataMAE, 'g')
plt.plot(xlim,real, 'b')
plt.scatter(jnp.linspace(-2,2),y)
```

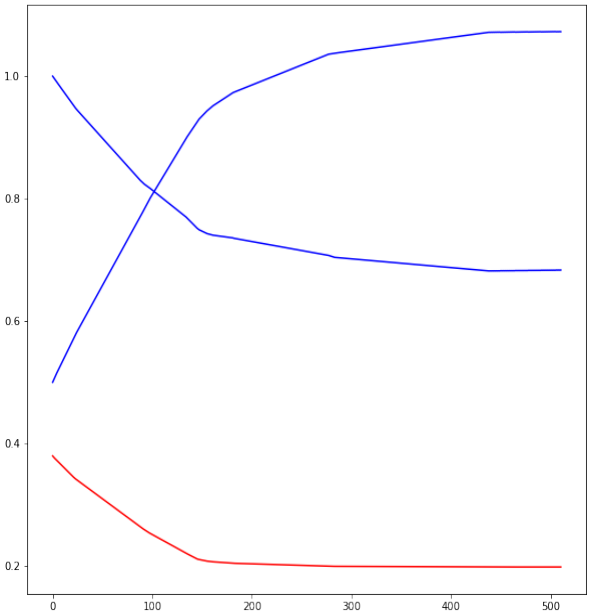


Se puede ver que los dos conjuntos de parámetros se ajustan bastante bien a la función original. Existen pequeñas diferencias que dependen de los ejemplos concretos que aparecen en diferentes puntos que hacen por ejemplo que la función obtenida usando el error absoluto (en verde) se acerque algo menos porque pesa menos errores más grandes.

Podemos representar también la evolución de los parámetros y de la función de error para cada caso. Podemos ver que el camino que sigue la optimización en ambos casos es ligeramente diferente.

El error cuadrático tiene una evolución más suave, ya que estamos explorando una función paraboloide, en el caso del error absoluto la evolución es lineal y podemos ver diferentes puntos de transición entre diferentes tendencias.


1.0
0.8
0.6
0.4
0.2
0
100
200
300
400
500
600
700


1.0
0.8
0.6
0.4
0.2
0
100
200
300
400
500