UNIVERSIDAD
DE GRANADA

Aprendizaje Automático
grado en ingeniería informática

MEMORIA PRÁCTICA 1

Autor
Vladislav Nikolov Vasilev

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

Curso 2018-2019

Índice

1. Ejercicio sobre la búsqueda iterativa de óptimos

Apartado 1

Implementar el algoritmo de gradiente descendente.

A continuación se muestra el código en Python implementado:

In [3]:
```python
def descent_gradient(initial_w, function, gradient, eta=0.01, threshold=None, iterations=100):
    """
    Función para el cálculo del gradiente descendente

    :param initial_w: Pesos iniciales
    :param function: Función a evaluar
    :param gradient: Función gradiente a utilizar
    :param eta: Valor de la tasa de aprendizaje (por defecto 0.01)
    :param threshold: Valor umbral con el que parar (por defecto None)
    :param iterations: Número máximo de iteraciones que tiene que hacer el bucle
                       (por defecto 100)

    :returns: Devuelve el peso final (w), el número de iteraciones que ha llevado
              conseguir llegar hasta éste, un array con todos los w y un array con
              los valores de w evaluados en function
    """

    w = np.copy(initial_w)                  # Se copia initial_w para evitar modificarlo
    iter = 0                                # Se inicializan las iteraciones a 0
    w_list = []                             # Se inicializa una lista vacía con los valors de w
    func_values_list = []                   # Se inicializa una lista vacía con los valors de la función

    w_list.append(w)                        # Añadir valor inicial de w
    func_values_list.append(function(*w))   # Añadir valor inicial de w evaluado en function

    # Se realiza el cálculo de la gradiente descendente mientras no se superen
    # el número máximo de iteraciones.
    while iter < iterations:
        iter += 1
        w = w - eta * gradient(*w)              # Actualización de w con los nuevos valores

        w_list.append(w)                        # Añadir nuevo w
        func_values_list.append(function(*w))   # Añadir nueva evaluación de w en function

        # Si se ha especificado un umbral en el que parar y se ha pasado
        # se sale del bucle
        if threshold and function(*w) < threshold:
            break

    return w, iter, np.array(w_list), np.array(func_values_list)
```

Se ha intentado que esta implementación sea lo más general posible para poder utilizarla en los ejercicios posteriores, parametrizando la función que recibe (parámetro **function**), la cuál puede ser tanto $f(x, y)$ como $E(u, v)$. Con este motivo, también se ha parametrizado el error con el que se quiere ajustar, ya que en un caso no será necesario utilizar un error como criterio de parada (de ahí que su valor por defecto sea **None**). Y, adicionalmente, se ha parametrizado el gradiente (parámetro **gradient**), para que también se pueda especificar a la hora de la llamada cuál se usará.

Apartado 2

Considerar la función $E(u, v) = (u^2e^v - 2v^2e^{-u})^2$. Usar gradiente descendente para encontrar un mínimo de esta funcón, comenzando desde el punto $(u, v) = (1, 1)$ y usanto una tasa de aprendizaje $\eta = 0{,}01$.

a) Calcular analíticamente y mostrar la expresión del gradiente de la función $E(u, v)$.

Para calcular el gradiente, vamos a calcular antes $\frac{\partial E}{\partial u}$ y $\frac{\partial E}{\partial v}$. Las derivadas, al aplicar la regla de la cadena, quedarían de la siguiente forma:

$$\frac{\partial E}{\partial u} = \frac{\partial}{\partial u}\Big((u^2e^v - 2v^2e^{-u})^2\Big) = 2(u^2e^v - 2v^2e^{-u})\frac{\partial(u^2e^v - 2v^2e^{-u})}{\partial u} = \\ = 2(u^2e^v - 2v^2e^{-u})(2ue^v + 2v^2e^{-u}) \tag{1}$$

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

Con esto, tenemos que la expresión del gradiente es la siguiente:

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial u} \\ \\ \frac{\partial E}{\partial v} \end{bmatrix} \tag{3}$$

$$\nabla E = \begin{bmatrix} 2(u^2e^v - 2v^2e^{-u})(2ue^v + 2v^2e^{-u}) \\ \\ 2(u^2e^v - 2v^2e^{-u})(u^2e^v - 4ve^{-u}) \end{bmatrix} \tag{4}$$

b) ¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de $E(u, v)$ inferior a 10^{-14}? (Usar flotantes de 64 bits)

```
In [5]: # Se fijan los parámetros que se van a usar en el cómputo de la gradiente descendente
        # (w inicial, num. iteraciones, valor mínimo)
        initial_w = np.array([1.0, 1.0])
        max_iter = 10000000000
        error = 1e-14

        w, it, w_array, func_val = descent_gradient(initial_w, E, gradient_E, threshold=error, iterations=max_iter)

        print ('Numero de iteraciones: ', it)
        print ('Coordenadas obtenidas: (', w[0], ', ', w[1],')')

Numero de iteraciones:  33
Coordenadas obtenidas: ( 0.6192076784506378 ,  0.9684482690100485 )
```

Figura 1: Cálculo del mínimo mediante el Gradiente Descendente para la función $E(u, v)$.

Como se puede ver en la figura anterior, donde también se incluye el código, el algoritmo tarda 33 iteraciones en encontrar por primera vez un valor de $E(u, v)$ inferior a 10^{-14}.

c) ¿En qué coordenadas (u, v) se alcanzó por primera vez un valor igual o menor a 10^{-14} en el apartado anterior?

Las coordenadas donde se alcanzó un valor inferior a 10^{-14} son $(0{,}619, 0{,}968)$ (redondeadas a 3 cifras decimales).

Apartado 3

Considerar ahora la función $f(x, y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)$.

a) Usar gradiente descendente para minimizar esta función. Usar como punto inicial $(x_0 = 0{,}1, y_0 = 0{,}1)$, tasa de aprendizaje $\eta = 0{,}01$ y un máximo de 50 iteraciones. Repetir el experimento pero usando $\eta = 0{,}1$, comentar las diferencias y su dependencia de η.

Antes de mostrar las gráficas, vamos a calcular el gradiente de la función $f(x, y)$. Primero, vamos a calcular $\frac{\partial f}{\partial x}$ y $\frac{\partial f}{\partial y}$:

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x}\Big(x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)\Big) = 2x + 4\pi\cos(2\pi x)\sin(2\pi y) \tag{5}$$

$$\frac{\partial f}{\partial y} = \frac{\partial}{\partial y}\Big(x^2 + 2y^2 + 2\sin(2\pi x)\sin(2\pi y)\Big) = 4y + 4\pi\sin(2\pi x)\cos(2\pi y) \tag{6}$$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{7}$$

$$\nabla f = \begin{bmatrix} 2x + 4\pi\cos(2\pi x)\sin(2\pi y) \\ \\ 4y + 4\pi\sin(2\pi x)\cos(2\pi y) \end{bmatrix} \tag{8}$$

Una vez calculada la expresión del gradiente, vamos a ejecutar el código y a realizar la comparación de los cálculos del gradiente al cambiar el valor de η:

In [6]:
```
# Se fijan los parámetros que se van a usar en el cómputo de la gradiente descendente
# en los dos casos
# w inicial, eta del segundo caso a estudiar y número máximo de iteraciones
initial_w = np.array([0.1, 0.1])
eta = 0.1
max_iter = 50

# Primer caso: w_inicial = (0.1, 0.1), eta 0.01, iteraciones = 50
w_1, it_1, w_array_1, func_val_1 = descent_gradient(initial_w, f, gradient_f, iterations=max_iter)

# Segundo caso: w_inicial = (0.1, 0.1), eta 0.1, iteraciones = 50
w_2, it_2, w_array_2, func_val_2 = descent_gradient(initial_w, f, gradient_f, eta=eta, iterations=max_iter)

# Mostrar por pantalla los resultados obtenidos
print('eta = 0.01')
print('Coordenadas obtenidas = ({}, {})'.format(w_1[0], w_1[1]))
print('Valor de la función = {}\n'.format(func_val_1[-1]))

print('eta = 0.1')
print('Coordenadas obtenidas = ({}, {})'.format(w_2[0], w_2[1]))
print('Valor de la función = {}'.format(func_val_2[-1]))
```

```
eta = 0.01
Coordenadas obtenidas = (0.24380496936478835, -0.23792582148617766)
Valor de la función = -1.8200785415471563

eta = 0.1
Coordenadas obtenidas = (0.10039167365942725, -1.0157510051441512)
Valor de la función = 1.9570333596159941
```

Figura 2: Valores de los óptimos para la función $f(x, y)$ cambiando η-

A continuación, se muestra el gráfico comparativo:

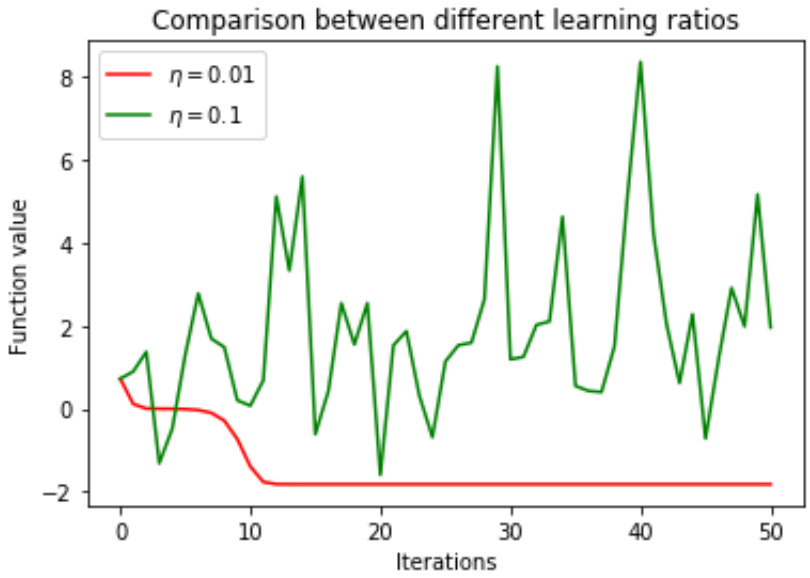

Figura 3: Comparativa gráfica de la Gradiente Descendente con diferentes valores de η.

Como se puede ver, después de 50 iteraciones, al utilizar un $\eta = 0{,}01$ se consigue converger a un óptimo local, mientras que al utilizar un $\eta = 0{,}1$ no se consigue.

En el primer caso, los valores de w van modificándose poco a poco con cada iteración, y por eso en el gráfico que puede ver que tiene una forma muy suavizada el cambio que sufre w entre iteración e iteración. En cambio, en el segundo caso, al tener un η mayor, se le da mucho peso al gradiente, y por tanto, los valores de w van modificándose muy bruscamente, como si estuviese oscilando entre la parte previa al mínimo y la posterior, sin llegar a converger en ningún momento.

Por tanto, el valor de η juega un factor clave en el algoritmo del descenso de gradiente. Si es muy bajo, los valores de w van descendiendo poco a poco, pero si el número de iteraciones son suficientes, se asegura llegar a un óptimo local. Sin embargo, si el η es muy grande, no se asegura en ningún momento que se pueda llegar al óptimo local, ya que puede ir saltando entre la parte anterior y la posterior al mínimo indefinidamente, llegando incluso a poder alejarse de éste.

b) Obtener el valor mínimo y los valores de las variables (x, y) en donde se alcanzan cuando el punto de inicio se fija: $(0{,}1, 0{,}1), (1, 1), (-0{,}5, -0{,}5), (-1, -1)$. Generar una tabla con los valores obtenidos.

A continuación se muestra una captura de pantalla con el código que permite generar los datos y la salida en formato tabla, donde se muestran los 4 puntos con sus coordenadas iniciales, finales y valor del punto final.

In [7]:
```
# Se fijan los parámetros que se van a usar en el cómputo de la gradiente descendente
# en los dos casos
# w inicial de cada caso y número máximo de iteraciones
initial_w_1 = np.array([0.1, 0.1])
initial_w_2 = np.array([1.0, 1.0])
initial_w_3 = np.array([-0.5, -0.5])
initial_w_4 = np.array([-1.0, -1.0])
max_iter = 50

# Cálculo del gradiente descendente para cada caso
w_1, it_1, w_array_1, func_val_1 = descent_gradient(initial_w_1, f, gradient_f, iterations=max_iter)
w_2, it_2, w_array_2, func_val_2 = descent_gradient(initial_w_2, f, gradient_f, iterations=max_iter)
w_3, it_3, w_array_3, func_val_3 = descent_gradient(initial_w_3, f, gradient_f, iterations=max_iter)
w_4, it_4, w_array_4, func_val_4 = descent_gradient(initial_w_4, f, gradient_f, iterations=max_iter)

# Mostrar por pantalla los resultados obtenidos usando pandas
# Crear una lista con los nombres de las columnas
column_header = ['x_0', 'y_0', 'x_f', 'y_f', 'Valor punto final']
row_header = ['Punto 1', 'Punto 2', 'Punto 3', 'Punto 4']

# Crear un array con los valores de cada fila
rows = np.array([[initial_w_1[0], initial_w_1[1], w_array_1[-1, 0], w_array_1[-1, 1], func_val_1[-1]],
                 [initial_w_2[0], initial_w_2[1], w_array_2[-1, 0], w_array_2[-1, 1], func_val_2[-1]],
                 [initial_w_3[0], initial_w_3[1], w_array_3[-1, 0], w_array_3[-1, 1], func_val_3[-1]],
                 [initial_w_4[0], initial_w_4[1], w_array_4[-1, 0], w_array_4[-1, 1], func_val_4[-1]]])

# Crear un nuevo DataFrame
df = pandas.DataFrame(rows, index=row_header, columns=column_header)

# Mostrarlo por pantalla
print(df)
```

```
         x_0  y_0       x_f       y_f  Valor punto final
Punto 1  0.1  0.1  0.243805 -0.237926          -1.820079
Punto 2  1.0  1.0  1.218070  0.712812           0.593269
Punto 3 -0.5 -0.5 -0.731377 -0.237855          -1.332481
Punto 4 -1.0 -1.0 -1.218070 -0.712812           0.593269
```

Apartado 4

¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el máximo global de una función arbitraria?

Encontrar el máximo global de una función arbitraria puede llegar a ser difícil y depende de una serie factores:

- **Forma de la función**: Si la función es convexa, como solo tiene un mínimo, es fácil encontrarlo con algoritmos como por ejemplo el Gradiente Descendente. Si la función tiene muchos óptimos locales o muchas curvaturas puede llegar a ser difícil dar con el óptimo global, y los algoritmos iterativos pueden llegar a quedarse en óptimos locales.
- **Punto de inicio**: El punto desde el que se empieza puede influir en el óptimo al que se llegue. En algunos casos, el punto inicial puede hacer que al aplicar algoritmos se llegue a un óptimo local y no se consiga salir de ahí. En otros casos, el punto puede estar más cerca del óptimo global, y por tanto se puede llegar a éste más fácilmente. Por tanto, hay que elegir con cuidado en qué punto se debería empezar.
- **El ratio de aprendizaje** (η): En algunos algoritmos, como por ejemplo el Gradiente Descendente y sus variantes, se utiliza un ratio de aprendizaje que permite dar más o menos peso al gradiente a la hora de acutalizar los valores de w. En caso de escoger un valor muy pequeño, la velocidad a la que se va a converger a un óptimo será más lenta, y por tanto se necesitarán más iteraciones. Un ratio muy elevado hará que sea muy difícil converger, ya que el valor de w irá pegando muchos saltos. Por tanto, un ratio de aprendizaje adecuado hará que se pueda llegar a un óptimo de mejor o peor forma. Dicho esto, ninguno de los dos garantiza que el óptimo al que se acabe llegando sea global, ya que perfectamente puede llegar a un óptimo local y quedarse ahí.

2. Ejercicio sobre Regresión Lineal

Este ejercicio ajusta modelos de regresión a vectores de características extraidos de imágenes de digitos manuscritos. En particular se extraen dos característcas concretas: el valor medio del nivel de gris y simetría del número respecto de su eje vertical. Solo se seleccionarán para este ejercicio las imágenes de los números 1 y 5.

Apartado 1

Estimar un modelo de regresión lineal a partir de los datos proporcionados de dichos números (Intensidad promedio, Simetria) usando tanto el algoritmo de la pseudoinversa como Gradiente descendente estocástico (SGD). Las etiquetas serán

$\{-1, 1\}$, una para cada vector de cada uno de los números. Pintar las soluciones obtenidas junto con los datos usados en el ajuste. Valorar la bondad del resultado usando E_{in} y E_{out} (para E_{out} calcular las predicciones usando los datos del fichero de test). (usar $Regress\_Lin(datos, label)$ como llamada para la función (opcional)).

Primero, vamos a observar las funciones implementadas para realizar los ajustes:

In [7]:
```
# Funcion para calcular el error
def Err(x, y, w):
    error = np.square(x.dot(w) - y.reshape(-1, 1))  # Calcular el error cuadrático para cada vector de caracterísi
    error = error.mean()                            # Calcular la media de los errors cuadráticos (matriz con una c

    return error

# Derivada de la función del error
def diff_Err(x,y,w):
    d_error = x.dot(w) - y.reshape(-1, 1)           # Calcular producto vectorial de x*w y restarle y
    d_error =  2 * np.mean(x * d_error, axis=0)     # Realizar la media del producto escalar de x*error y la media

    d_error = d_error.reshape(-1, 1)                # Cambiar la forma para que tenga 3 filas y 1 columna

    return d_error
```

Figura 4: Función de error y derivada de ésta para el gradiente.

In [10]:
```
# Pseudoinversa
def pseudoinverse(X, y):
    """
    Función para el cálculo de pesos mediante el algoritmo de la pseudoderivada

    :param X: Matriz que contiene las caracterísiticas
    :param y: Matriz que contiene las etiquetas relacionadas a las características

    :returns w: Pesos calculados mediante ecuaciones normales
    """

    X_transpose = X.transpose()                     # Guardamos la transpuesta de X
    y_transpose = y.reshape(-1, 1)                  # Convertimos y en una matriz columna (1 fila con n columnas)

    # Aplicamos el algoritmo para calcular la pseudoinversa
    w = np.linalg.inv(X_transpose.dot(X))
    w = w.dot(X_transpose)

    # Hacemos el producto de matrices de la pseudoinversa y la matriz columna y
    w = w.dot(y_transpose)

    return w
```

Figura 5: Implementación del algoritmo de la pseudoinversa.

In [8]:
```python
# Gradiente Descendente Estocastico
def sgd(X, y, eta, M=64, iterations=200):
    """
    Función para calcular el Gradiente Descendente Estocástico.
    Selecciona minibatches aleatorios de tamaño M de la muestra original
    y ajusta en un número de iteraciones los pesos.

    :param X: Muestra de entrenamiento
    :param y: Vector de etiquetas
    :param eta: Ratio de aprendizaje
    :param M: Tamaño de un minibatch (64 por defecto)
    :param iterations: Número máximo de iteraciones

    :return w: Pesos ajustados
    """

    # Crear un nuevo vector de pesos inicializado a 0, establecer el número de iteraciones
    # inicial y obtener el número de elementos (N)
    w = np.zeros((3, 1), np.float64)
    N = X.shape[0]
    iter = 0

    # Mientras el número de iteraciones sea menor al máximo, obtener un minibatch
    # de tamaño M con valores aleatorios de X y ajustar los pesos con estos valores
    while iter < iterations:
        iter += 1

        # Escoger valores aleatorios de índices sin repeticiones y obtener los elementos
        index = np.random.choice(N, M, replace=False)
        minibatch_x = X[index]
        minibatch_y = y[index]

        # Actualizar w
        w = w - eta * diff_Err(minibatch_x, minibatch_y, w)

    return w
```

Figura 6: Implementación del Gradiente Descendente Estocástico.

Apartado 2

En este apartado exploramos como se transforman los errores E_{in} y E_{out} cuando aumentamos la complejidad del modelo lineal usado. Ahora hacemos uso de la función $simula\_unif(N, 2, size)$ que nos devuelve N coordenadas 2D de puntos uniformemente muestreados dentro del cuadrado definido por $[-size, size] \times [-size, size]$

- EXPERIMENTO

  a) Generar una muestra de entrenamiento de $N = 1000$ puntos en el cuadrado $X = [-1, 1] \times [-1, 1]$. Pintar el mapa de puntos 2D. (ver función de ayuda)

  b) Consideremos la función $f(x_1, x_2) = \text{sign}((x_1 - 0{,}2)^2 + x_2^2 - 0{,}6)$ que usaremos para asignar una etiqueta a cada punto de la muestra anterior. Introducimos ruido sobre las etiquetas cambiando aleatoriamente el signo de un 10 % de las mismas. Pintar el mapa de etiquetas obtenido.

  c) Usando como vector de características $(1, x_1, x_2)$ ajustar un modelo de regresión lineal al conjunto de datos generado y estimar los pesos w. Estimar el error de ajuste E_{in} usando Gradiente Descendente Estocástico (SGD).

d) Ejecutar todo el experimento definido por (a)-(c) 1000 veces (generamos 1000 muestras diferentes) y

- Calcular el valor medio de los errores E_{in} de las 1000 muestras.
- Generar 1000 puntos nuevos por cada iteración y calcular con ellos el valor de E_{out} en dicha iteración. Calcular el valor medio de E_{out} en todas las iteraciones.

e) Valore que tan bueno considera que es el ajuste con este modelo lineal a la vista de los valores medios obtenidos de E_{in} y E_{out}.

3. Bonus

Apartado 1

Método de Newton. Implementar el algoritmo de minimización de Newton y aplicarlo a la función $f(x, y)$ dada en el ejercicio 3. Desarrolle los mismos experimentos usando los mismos puntos de inicio.

- Generar un gráfico de como desciende el valor de la función con las iteraciones.
- Extraer conclusiones sobre las conductas de los algoritmos comparando la curva de decrecimiento de la función calculada en el apartado anterior y la correspondiente obtenida con el gradiente descendente.

Referencias

[1] Texto referencia
`https://url.referencia.com`