Trabajo 2 Aprendizaje Automático

Antonio Mannuel Milán Jiménez

27 de abril de 2017

Ejercicio 1: Gradiente Descendente

Apartado a

En este apartado tenemos que utilizar el gradiente descendiente para encontrar el mínimo de la función:

$$e = (w[1]^2 * e^{w[2]} - 2 * w[2]^2 * e^{-w[1]})^2$$

.

Para ello hemos derivado esta función respecto a u:

$$4 * e^{-2*u} * (u^2 * e^{u+v} - 2 * v^2) * (u * e^{u+v} + v^2)$$

Y respecto a v:

$$2 * e^{-2*u} * (u^2 * e^{u+v} - 4 * v) * (u^2 * e^{u+v} - 2 * v^2)$$

Estas dos derivadas están implementadas en las funciones "derivada_u" y "derivada_v" respectivamente. Así, el gradiente recibe una w, realiza las derivadas con esta w y devuelve los dos resultados. Ya en el algoritmo del gradiente descendente vamos calculando el gradiente, que lo utilizamos junto con una tasa para actualizar nuestro vector w:

w= w+ tasa*mi_vector

En el momento en el que para nuestra w el error es menor a 10^-4, termina el algoritmo. Si ejecutamos el algoritmo:

```
## [1] 24190
```

Vemos que hemos necesitado 24190 iteraciones hasta que hemos conseguido que el error sea menor a 10^-4.

```
## [1]  11.67841 -24.28938
```

Y esta es la w con la que hemos conseguido alcanzar dicho error, u= 11.678 y v= -24.0289

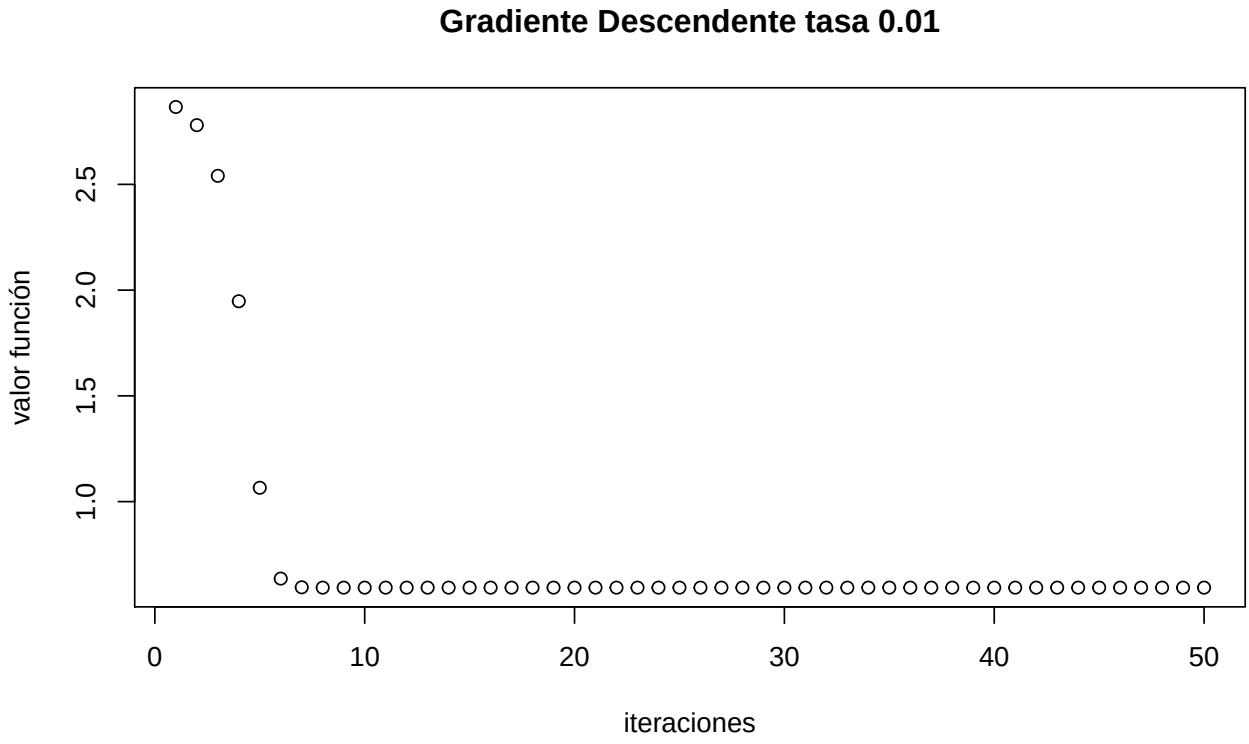
Apartado b

Para este apartado se nos proponía utilizar el gradiente descendente con una nueva función:

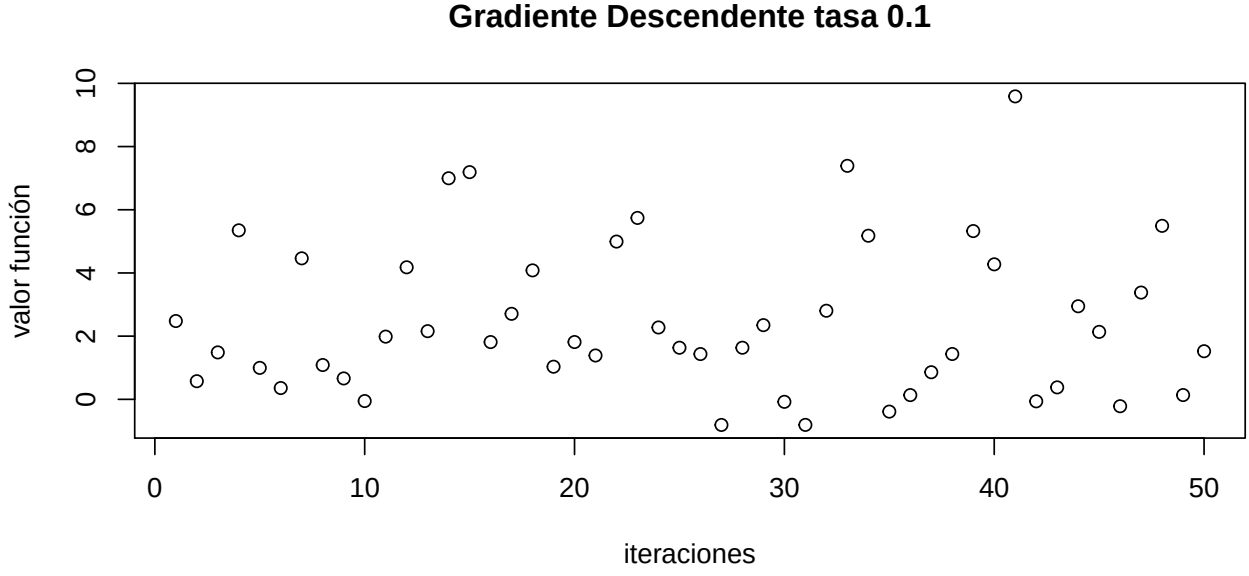
$$(x-2)^2 + 2 * (y-2)^2 + 2 * sin(2 * pi * x) * sin(2 * pi * y)$$

cambiando la tasa de aprendizaje y también el punto de inicio.

Esta es la gráfica que obtenemos con el punto de inicio en 1,1 y una tasa de aprendizaje de 0.01:



Sin embargo, si ahora cambiamos la tasa de aprendizaje a 0.1 obtenemos esta gráfica:



Las diferencias son claras. Con la tasa de aprendizaje de 0.01 se realiza un decrecimiento "controlado" hasta que al final llega a un punto cercano al mínimo donde la diferencia entre cada iteración es mínima ya que el cambio entre w es muy pequeño y vemos como llanea la función. En cambio, al utilizar una tasa de 0.1 vemos como los "saltos" entre cada iteración son mucho mayores y eso hace que esté continuando saltando entre puntos altos y bajos. Como consecuencia, no consigue estabilizarse, llanear y encontrar el mínimo.

Tambien hemos probado a variar el punto de inicio para ver si obtenemos diferentes resultados. Esta es la tabla con los diferentes resultados con el valor mínimo obtenido y las "x" e "y" que lo han obtenido:

```
##      x_ini y_ini     x_min    y_min     minimo
## [1,]   2.1   2.1 2.2438050 1.762074 -1.8200785
## [2,]   3.0   3.0 3.2180703 2.712812  0.5932694
## [3,]   1.5   1.5 1.2686225 1.762145 -1.3324811
## [4,]   1.0   1.0 0.7819297 1.287188  0.5932694
```

Comentado la tabla vemos que al empezar en (1,1) o (3,3) obtenemos el mismo mínimo, sin embargo, para (2.1,2.1) y (1.5,1.5) conseguimos acercarnos mucho más al mínimo.

Apartado c

Finalmente como conclusión sobre la dificultad de encontrar el mínimo tenemos una doble dificultad. Por una parte el punto de inicio, si bien los 4 puntos han tendido hacia el mínimo, 2 ellos se han acercado mucho más. Por lo tanto deberíamos probar siempre varios puntos de inicio puesto que puede variar mucho en el punto mínimo que encontremos. El otro punto de dificultad es la tasa de dificultad. Observando ambas gráficas parecería claro que es mejor utilizar una tasa pequeña para evitar esos saltos entre iteraciones, no obstante, como consecuencia tardará mucho más en encontrar el mínimo si la función llanea varias veces pues el avance será muy lento.

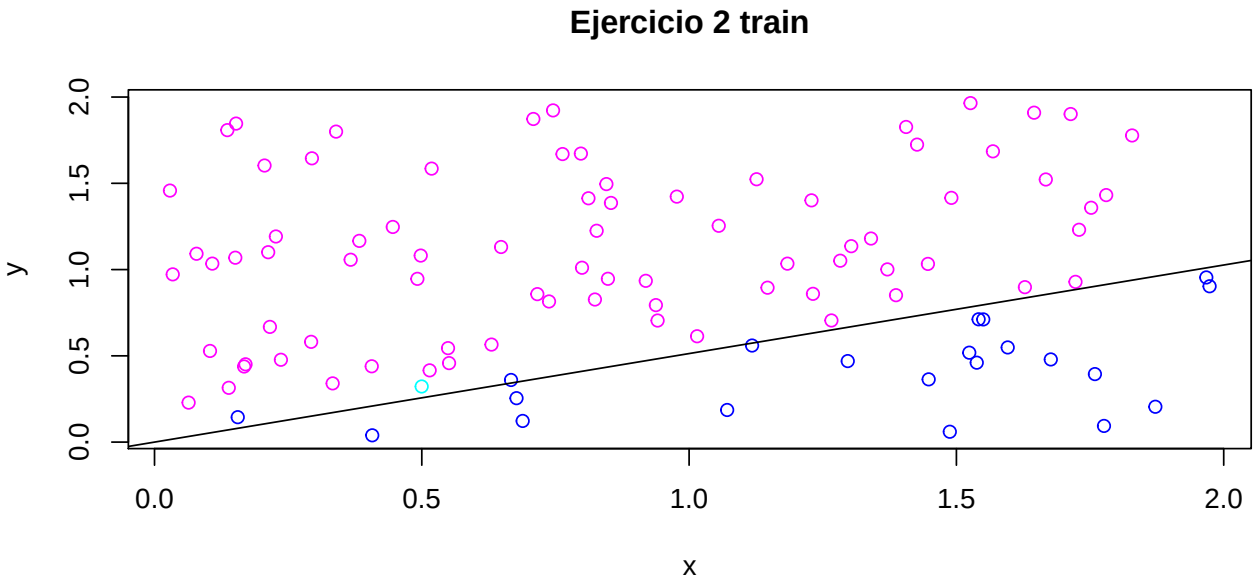
Ejercicio 2: Regresión Logística

Apartado a

Para implementar la regresión logística hemos utlizado algunas funciones que vimos en la anterior práctica como "simula_unif" o "simula_recta". También hemos cogido de la anterior práctica la función que a partir de una recta y unos datos, generaba etiquetas a esos datos, y la función que a partir de la w, los datos y las etiquetas, calculaba el error en la clasificación. Hemos implementado la función "gradiente Estocastico" que calcula el gradiente para un punto mediante la expresión: > r=-(etiquetayvectorCaractx)/(1 + exp(etiquetay(wvectorCaractx)))

El algoritmo de la regresión logística en cada época recorre todos los datos y sobre cada uno calcula el gradiente estocástico con el que actualiza la w, tal y como hacíamos en el ejercicio 1. Además, antes de cada época se realiza un "sample" para que en cada época se recorran los datos de forma aleatoria. Al final de cada época, se calcula la distancia entre la w anterior y la nueva w que acabamos de obtener. El algoritmo termina cuando esta distancia es menor a 0.01.

Ejecutandolo ahora para unos datos de "training" obtenemos la gráfica de clasificación:



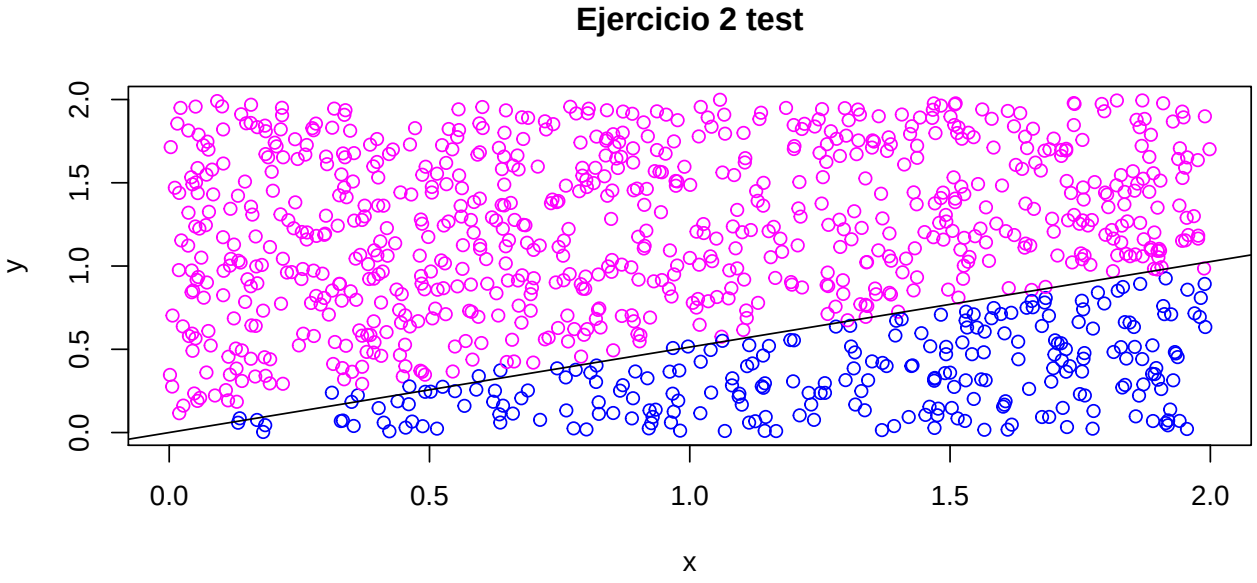
Y este es el error que hemos obtenido:

```
## [1] 3
```

Viendo la gráfica y el error de training vemos que se consigue un gran resultado al utilizar regresión logística y gradiente estocástico con sólo 3 puntos mal clasificados.

Apartado b

Ahora vamos a utilizar un conjunto de datos de 1000 puntos que lo trataremos como el conjunto de test, para ver el comportamiento de la regresión logística con unos datos que no se han trabajado sobre ellos. Esta es la gráfica para los datos de test:




```
## [1] 1.3
```

En la anterior práctica habíamos visto que el error de clasificación en los datos de test era mayor que en los de training, lógico teniendo en cuenta que la w se había calculado y ajustado trabajando sólo sobre los datos de training. Sin embargo, al haber incluido ahora el regresión logística con gradiente estocástico, obtenemos

un mejor resultado en el test con tan solo un 1,3% de error. Podríamos decir que este algoritmo no se ajusta tanto a los datos de training y eso le permite obtener mejores resultados para el test.

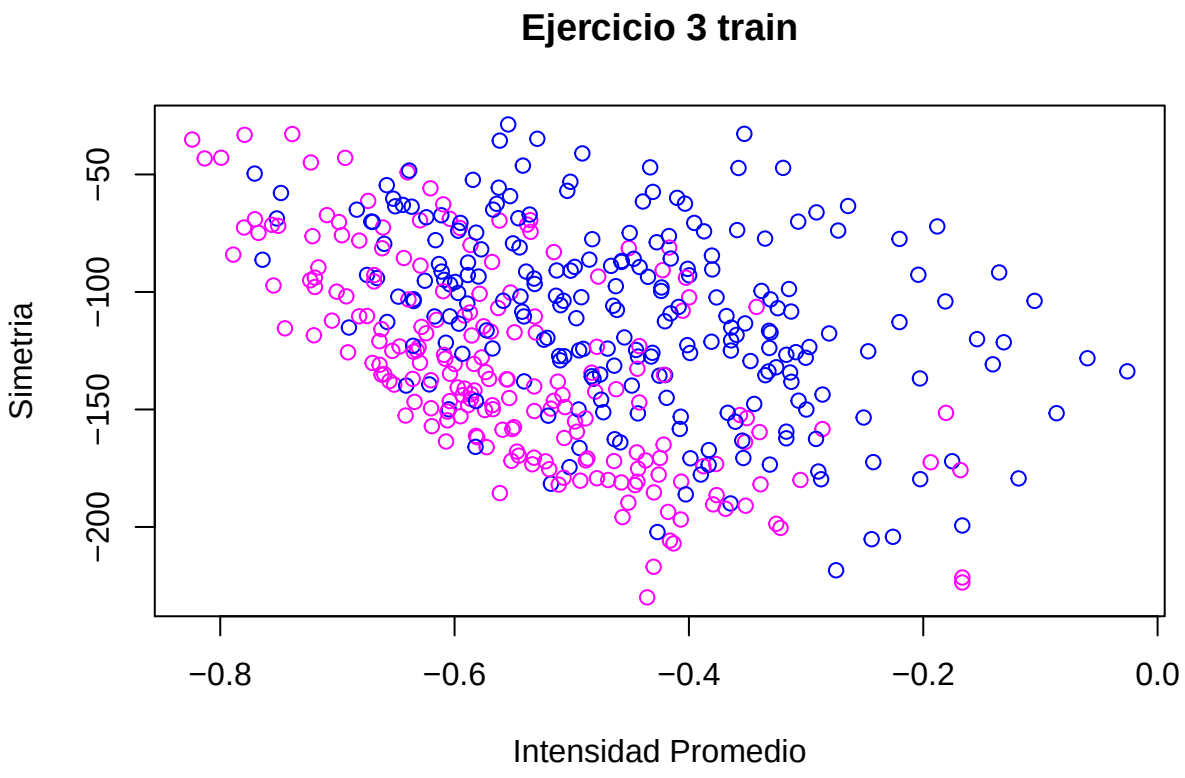
Ejercicio 3 Clasificación de Dígitos

Apartado a

En este ejercicio vamos a clasificar los dígitos en 4 y 8. Para ello, vamos a necesitar las matrices con los datos de media y simetría de los diferentes dígitos, además de las etiquetas de cada dígito de si es un 4 o un 8 para saber si estamos haciendo bien la clasificación y calcular los errores. Una vez con los datos de simetría y media, aplicaremos Regresión lineal y PLA-Pocket para obtener un vector de w que construirá una recta para clasificar futuros dígitos en 4 u 8.

Apartado b

En este ejercicio de clasificación de datos hemos utilizado funciones utilizadas en la anterior práctica para tratar las muestras de los dígitos 4 y 8, tales como obtener las imagenes a partir de los datos o la de calcular la matriz de media y simetría. Aquí podemos ver la gráfica de los datos:



Viendo a priori los datos vemos que va a ser dificil obtener una buena clasificación con un modelo lineal pues estan muy entremezclados los datos de ambos dígitos.

Primero utilizamos la regresión lineal sobre los datos para obtener un vector w inicial sobre el que empezar el PLA-Pocket. La regresión lineal es tal y como se hizo en la anterior práctica donde a partir de la pseudo_inversa de los datos y de las etiquetas, construimos un vector de pesos w. Este es el primer error para dentro de la muestra que obtenemos al realizar la regresión lineal:

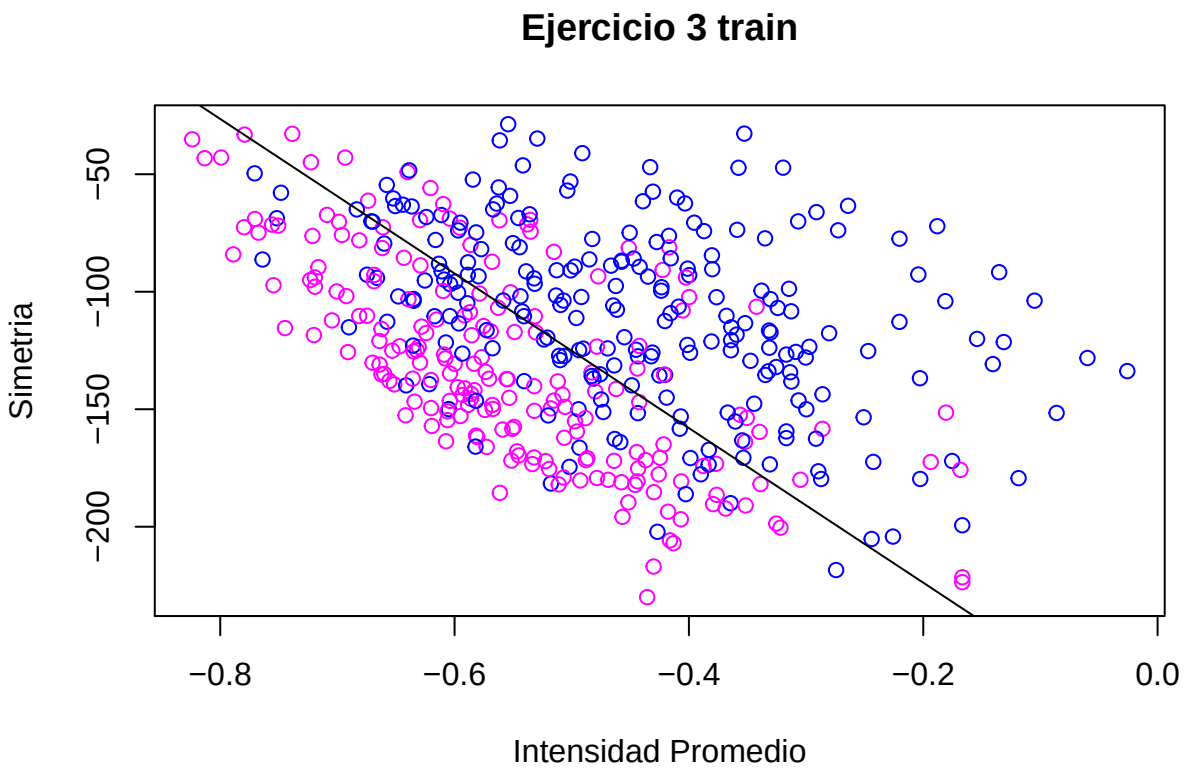
```
## [1] 24.76852
```

Como era de esperar el error es elevado, cerca del 25%. Como habíamos comentado anteriormente, viendo ya los datos y lo entremezclados que estaban, iba a ser dificil obtener una buena clasificación con un modelo lineal.

Continuando con el ejercicio vamos a aplicar sobre esta w el PLA-Pocket como mejora y valorar los resultados. El algoritmo PLA-Pocket funciona igual que el PLA de la anterior práctica salvo que ahora se encarga de guardar cúal ha sido la mejor w (en función del error obtenido) hasta ahora. Así, a diferencia del PLA que nos devolvía siempre la última w calculada, en el PLA-Pocket se nos devolverá la mejor w que se haya encontrado, independientemente de la iteración. Este es el error obtenido:

```
## [1] 24.76852
```

Y esta es la gráfica con la recta obtenida de la w calculada:

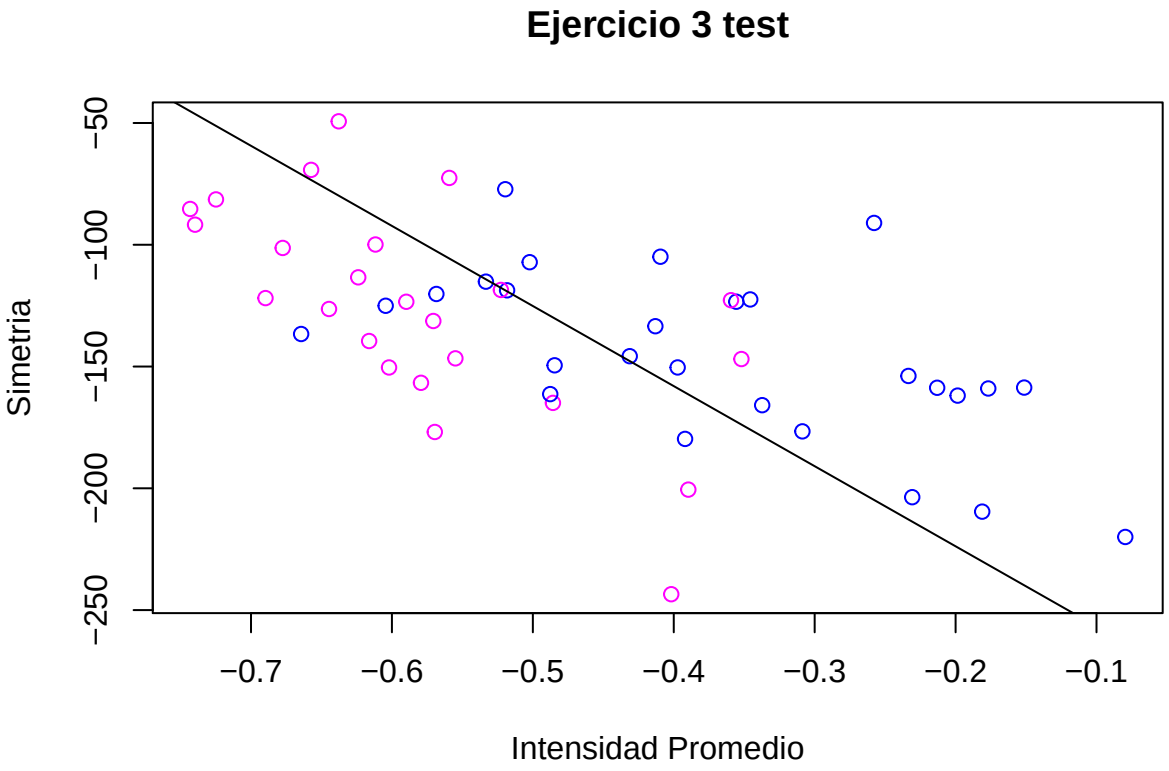


Vemos que aunque hemos aplicado el PLA-Pocket, el error es exactemente el mismo que antes de aplicarlo. Esto se debe a que con la regresión lineal hemos obtenido una w suficientemente "buena" que el PLA-Pocket no ha sido capaz de mejorar en todas sus iteraciones.

Ahora vamos a utilizar los datos de test para comprobar como de bien se comporta la recta calculada con unos datos fuera de la muestra. Este es el error fuera de la muestra:

```
## [1] 23.52941
```

Y está es la gráfica de la recta junto con los datos de test:



Vemos que el error para test es un poco mejor pero sigue siendo malo con un 23% de errores. Como hemos comentado en el anterior ejercicio, parece que la w no se ha sobreajustado a los datos de training y eso ha permitido obtener unos resultados similares, incluso mejores, que los de training y no los ha empeorado.

Cálculo de cotas

Además se nos pedía que calculasemos las cotas para Eout. Primero vamos a calcular la cota para Eout sobre Ein. Tomando Ein como cota para Eout tenemos la ecuacion:

$$eOut(g) <= eIn(g) + \sqrt{(\frac{8}{N} * ln(\frac{4 * 2N^3 + 1}{0.05})}$$

Si sustituimos y despejamos tenemos que eOut(g) <= eIn(g) + 0.67 como cota superior. Esta sería la cota:

```
## [1] 25.43852
```

Ahora vamos a calcular otra cota superior para Eout pero esta vez sobre Etest. Tomando P(|eTest(g) - eOut(g)| > exilon) como 0.05, podemos despejar el exilon de la ecuación

$$0.05 <= 2 * e^{(-2N * exilon^2)}$$

. Despejándolo obtenemos que exilon=0.19. Con este dato, utilizaríamos eTest como cota superior para eOut, eTest(g) + exilon > eOut. Aquí tenemos la cota:

```
## [1] 23.71941
```

Comparando ambas cotas podemos decir que ambas son bastantes estrictas pues están muy cerca de los resultados obtenidos en el ejercicio. Es cierto que la calculada con errorIn, 25.43 es un poco más relajada pues el errorIn ha sido más elevado que el de test y ha hecho sea tambien algo mayor. Sin embargo, la calculada con errorTest es muy estricta y sólo esta dejando un 0.19% de margen.

Ejercicio 4 Regularización en la selección de modelos

Apartado a

Para este ejercicio creamos unos datos y unas etiquetas a partir de "rnorm" y de la función simula_gauss que vimos en la anterior práctica. Sobre los datos de tamaño 113, creamos 10 particiones sobre las que haremos validación cruzada. Así, iremos tomando una partición como los datos de test, y las otras 9 como los datos de training. A continuación, obtendremos la w_reg con la siguiente expresión:

w_reg=pseudo_inversa(datos_train,0.05/113)t(datos_train)etiquetas_train

La función de la pseudo_inversa es la misma que utilizábamos en la anterior práctica salvo que ahora sumamos una matriz identidad y multiplicamos por una lambda tal y como se muestra a continuación:

aux=t(X)X + diag(ncol(X))lambda

Una vez que tenemos la w_reg, tenemos que calcular el error a partir de dicha w para la partición que habíamos dejado como test. El calculo del error se hace sobre un bucle que itera sobre los datos y etiquetas de test, y va calculando la expresión:

aux=aux + (w datos[i]-etiquetas[i])^2

En el ejercicio se nos pedia que concretamente obtuviesemos el error para la primera partición, para la segunda y la media de todos los errores de las particiones, Ecv. Este proceso se realiza 1000 veces, generando datos nuevos para cada iteración y haciendo la media de los resultados. Estos son los datos obtenidos que analizaremos en los siguientes apartados:

```
##           [,1]
## [1,] 0.2584373
```

```
##          [,1]
## [1,] 0.258915
```

```
##           [,1]
## [1,] 0.2588512
```

Apartado b

Comparando e1 con e2 y Ecv vemos que e1 ha sido un poco superior. Esto se debe a que su conjunto de datos era algo superior, 13, y tiene una probabilidad mayor de contener un error en su partición.

Apartado c

Como hemos comentado en el anterior apartado, el que la partición de e1 haya sido mayor al resto, ha hecho que aumenta la posibilidad de que alguno de esos 13 datos sea un error, de ahí que la media de error en la partición 1 haya sido algo mayor.

Apartado d

Como conclusión, hemos visto que la regularización weight decay presenta unos muy buenos resultados con tan solo 0.25% de error, unos resultados mucho mejores que cualquier otro algoritmo que hayamos visto anteriormente. Además, para los buenos resultados que presenta su ejecución es bastante rápida pues realizando la ejecución 1000 veces tarda escasos segundos.

Bonus

Ejercicio 1 Coordenada descendente

Para este ejercicio utilizamos la misma función del ejercicio 1.1a y una variante del gradiente descendente que hemos utilizado ya que, ahora primero solo actualizamos la w con la u, y a continuación actualizamos la w solo con la v. De esta forma, la w se mueve sólo en una dirección en cada paso.

En el ejercicio se nos pide que mostremos el error que se alcanza en la iteración 15 con una tasa de aprendizaje de 0.1. Sin embargo, rapidamente aumenta el error hasta que en la tercera iteración el error es tan grande que se hace indeterminado. No obstante, si cambiamos la tasa a una mucho más pequeña, 0.001, vemos que el error sí alcanza valores más lógicos. Aquí tenemos los errores para una tasa de 0.1 y 0.001 respectivamente:

```
## [1] NaN
```

```
## [1] 0.5845682
```

Apartado b

Se nos pide que hagamos una comparación para el gradiente descendente asi que aquí tenemos los errores alcanzados por el gradiente descendente en la iteración 15 para una tasa de 0.1 y 0.001 respectivamente:

```
## [1] 9.87299
```

```
## [1] 0.7854532
```

Comparando los errores vemos que para una tasa de 0.1 parece obvio que es mejor utilizar el gradiente descendente que la coordenada descendente puesto que está alcanza valores tan altos que se hace indeterminada. Si ahora trabajamos con una tasa de 0.01 destacamos 2 puntos importantes. El primero es que con esta tasa sí es mejor utilizar coordenada descendente ya que presenta mejores resultados, no obstante, el gradiente descendente también tiene un buen comportamiento acercandose al mínimo.

Como segundo punto, vemos que aunque para la tasa de 0.1 el gradiente descendente haya tenido un mejor resultado que la coordenada descendiente, no deja de ser un mal resultado puesta que vemos que a 9.87 está muy lejos del mínimo. Esto se deberá como se ha comentado en anteriores ejercicios a que está realizando saltos demasiados grandes entre las iteraciones y no consigue estabilizarse cerca del mínimo.

Ejercicio 2 Método de Newton

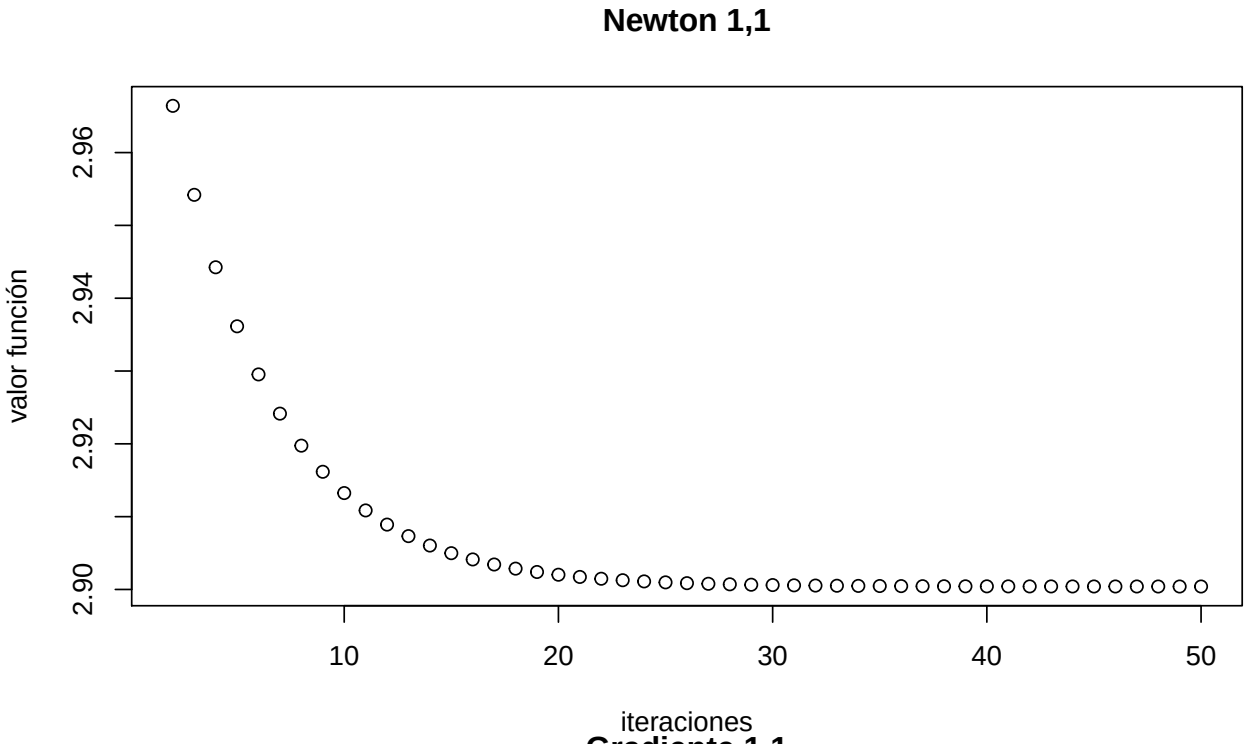
En el método de Newton, el calculo de la nueva w se hace:

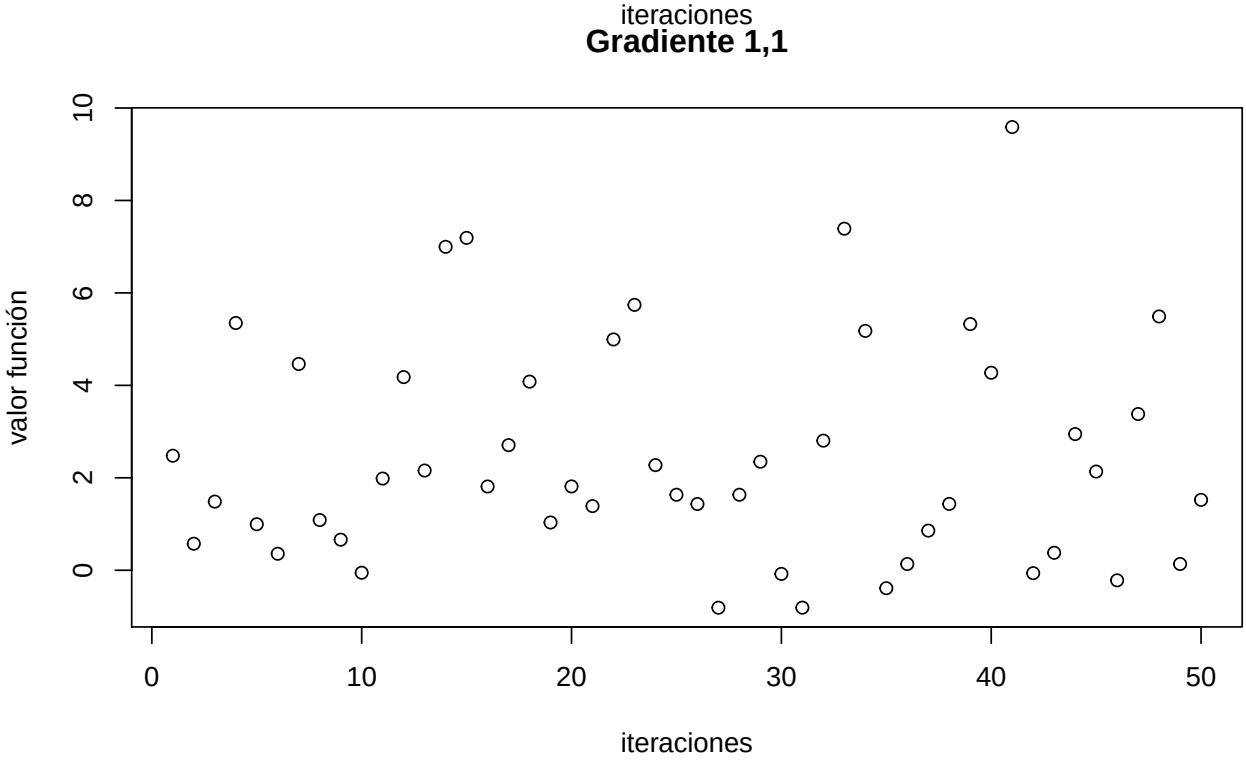
w= w+ tasa incremento_w

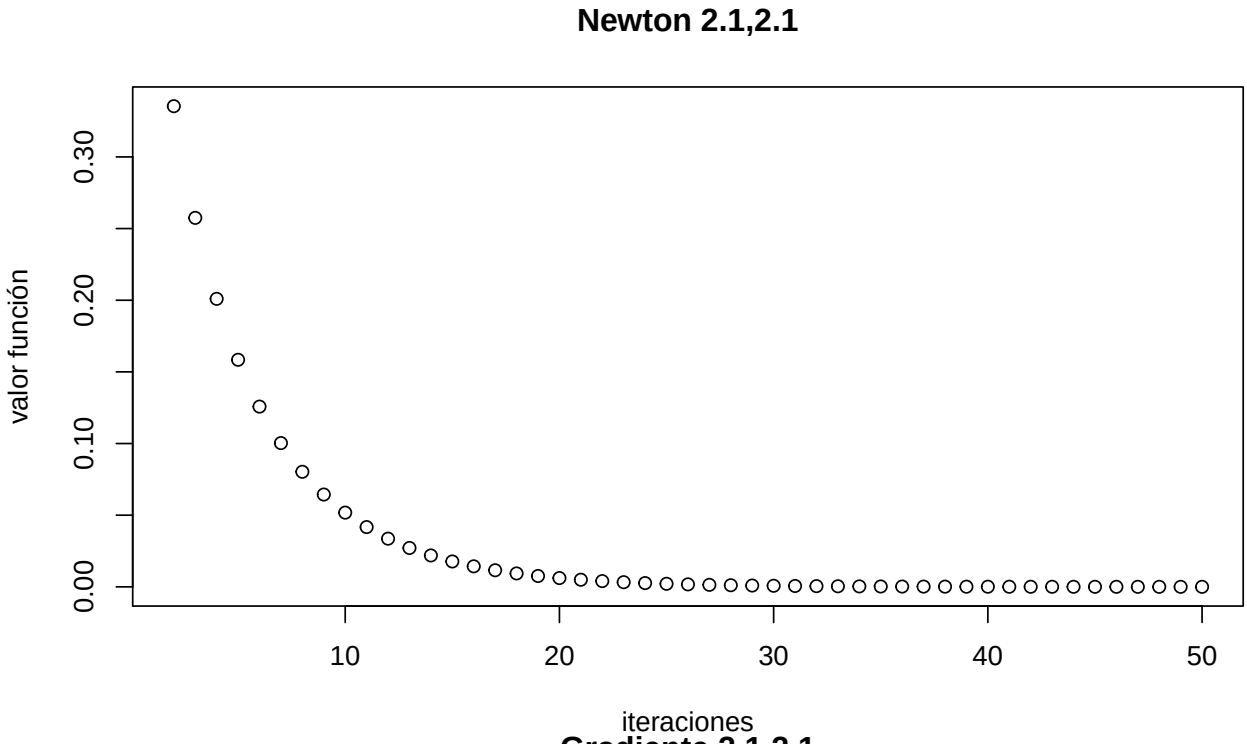
De forma similar al gradiente descendente salvo que ahora tenemos que calcular el incremento de w de otra forma. Este incremento de w se calcula multiplicando la inversa de la matriz jacobiana por el gradiente de w (tal y como se hacía en el ejercicio 1). Para esta matriz jacobiana se ha necesitado calcular: doble derivada respecto a “x”, derivada respecto a “x” y a continuación derivada respecto a “y”, derivada respecto a “y” y a

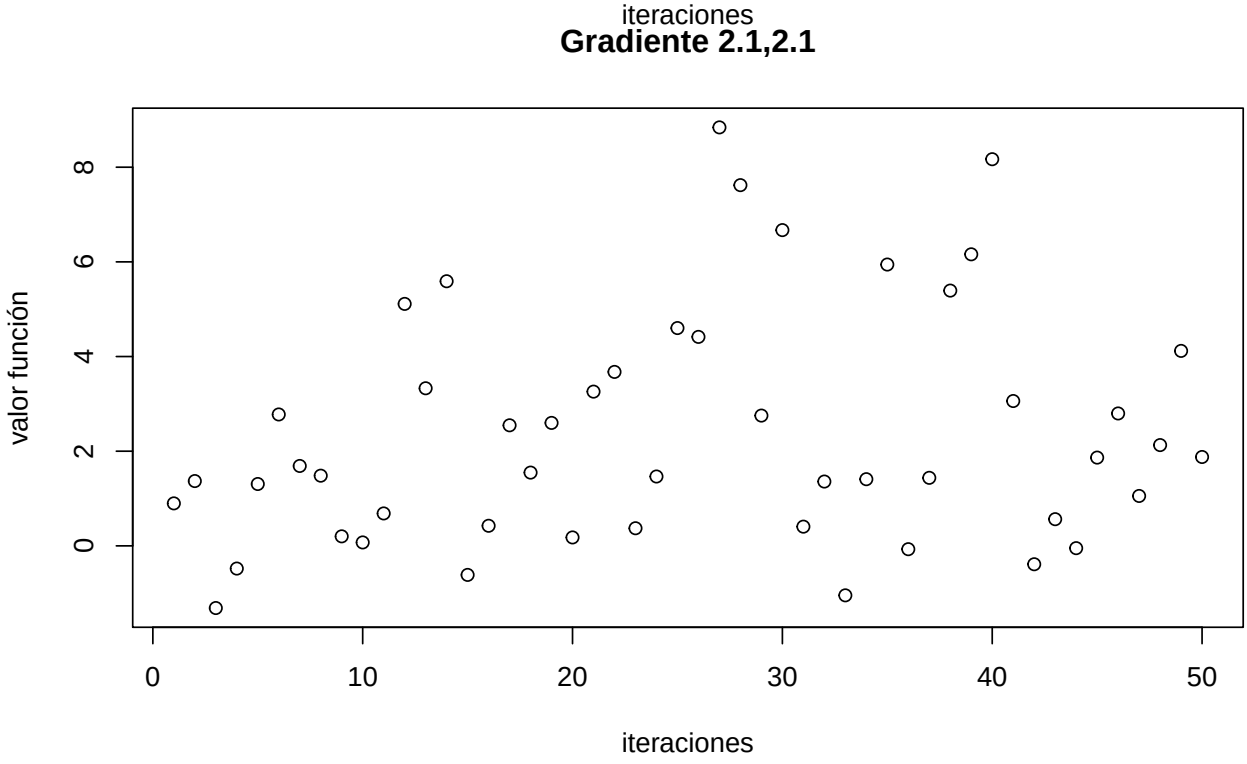
continuación derivada respecto a “x”, y doble derivada respecto a “y”. Así, se irá actualizando la w a lo largo de las iteraciones que marquemos.

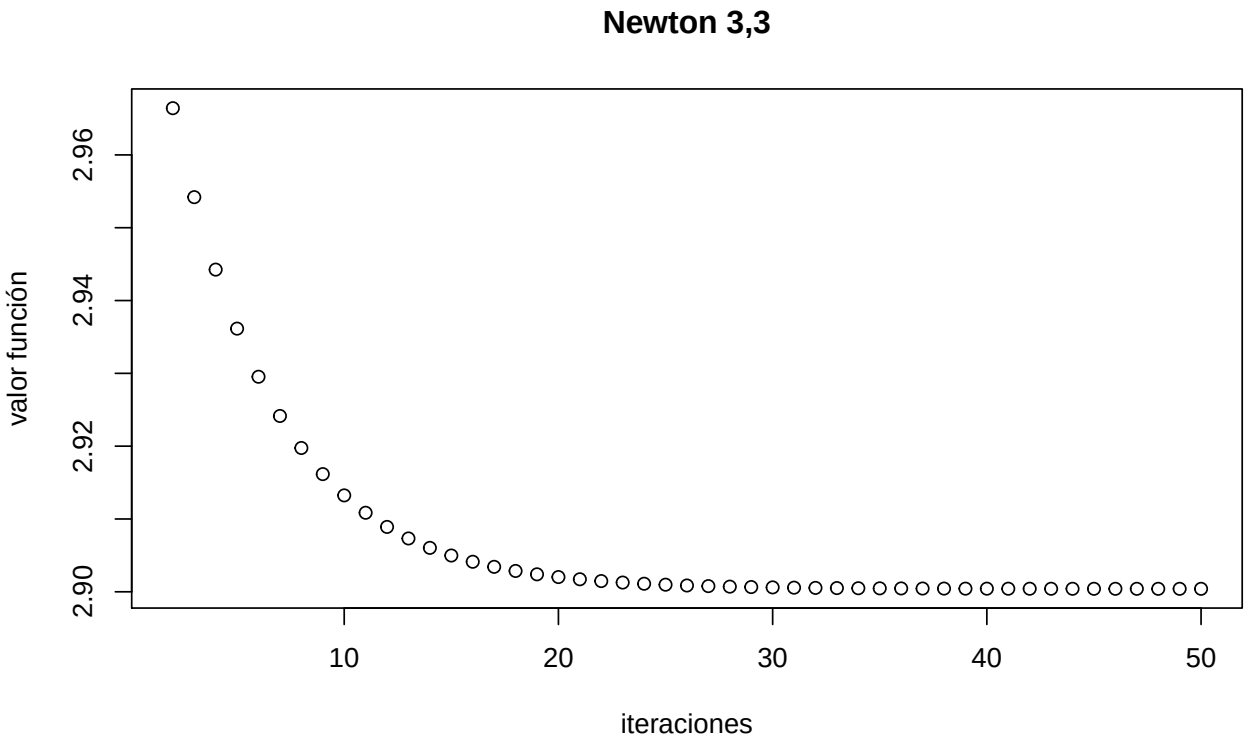
En el ejercicio se nos pide que comparemos la gráfica del valor de la función con la obtenida en el gradiente descendente del ejercicio 1.b. Además, como veremos que dependen los valores obtenidos en función del punto inicial escogido, haremos una comparación entre el método de Newton y el gradiente descendente para el mismo punto de inicio:

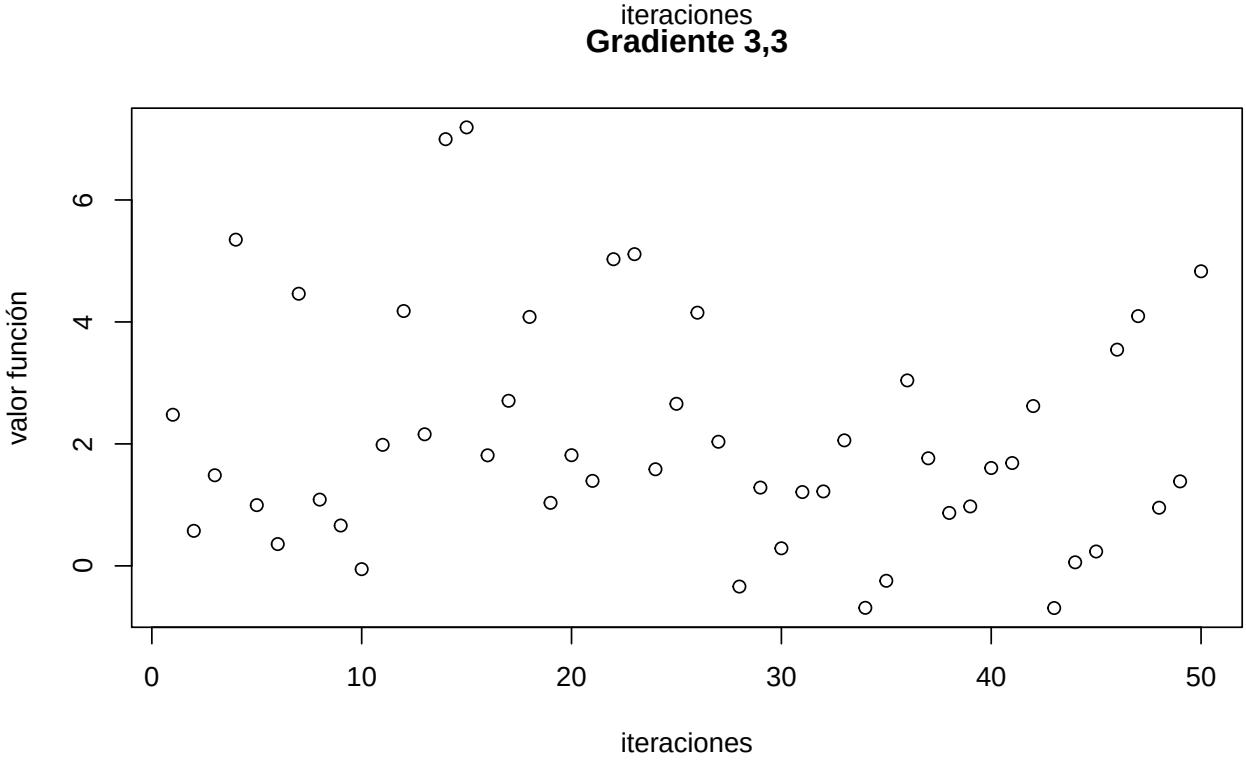


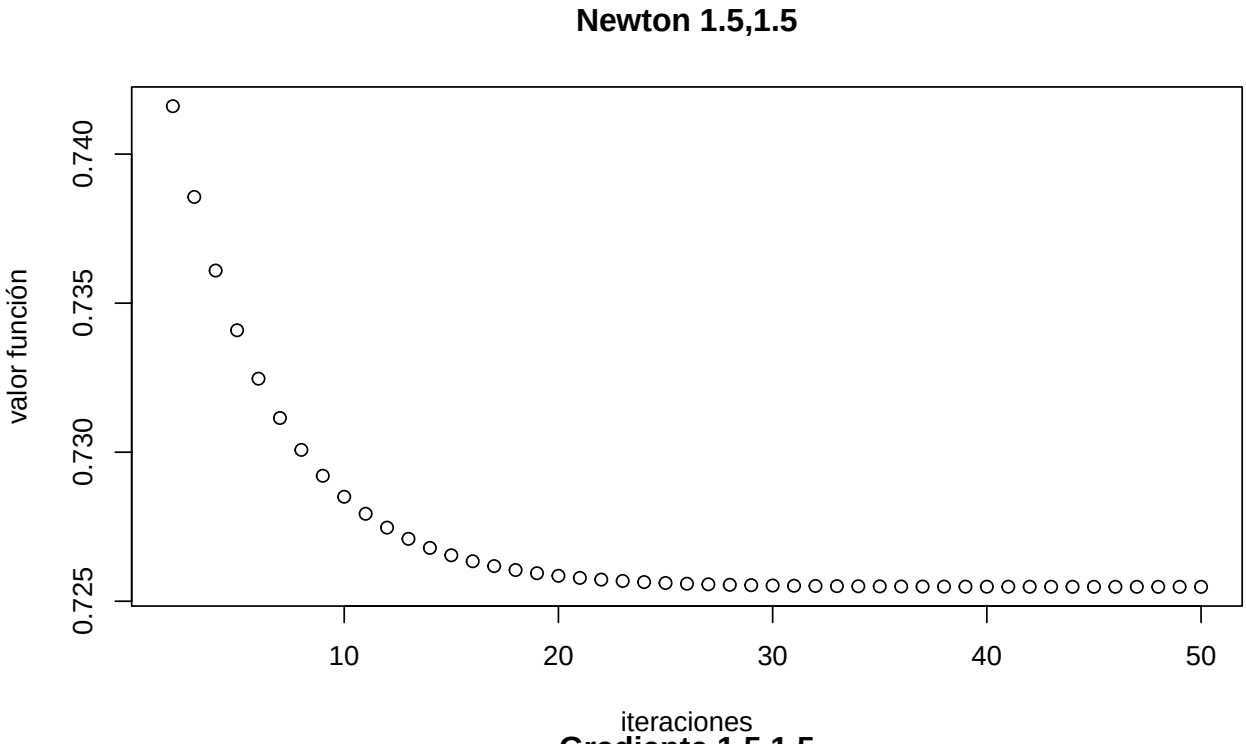


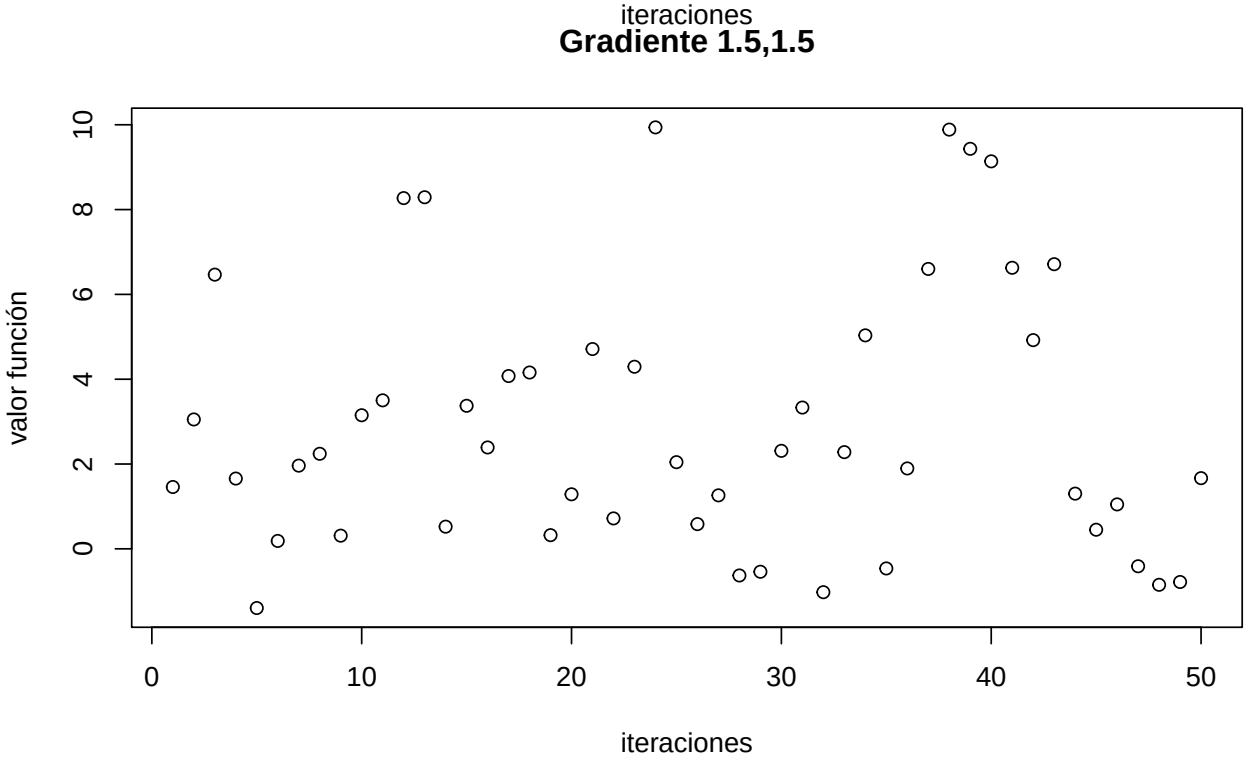

Newton 2.1,2.1
valor función
0.00
0.10
0.20
0.30
10
20
30
40
50
iteraciones


Gradiente 2.1,2.1
valor función
0
2
4
6
8
0
10
20
30
40
50
iteraciones


Newton 3,3
valor función
2.90
2.92
2.94
2.96
10
20
30
40
50
iteraciones


Gradiente 3,3
valor función
0
2
4
6
0
10
20
30
40
50
iteraciones





A partir de estas gráficas podemos extraer 2 grandes conclusiones.

La primera conclusión está relacionada con la tasa de aprendizaje. Tanto para Newton como para el gradiente descendente se ha utilizado una tasa de aprendizaje de 0.1, las diferencias son claras. Con una tasa de 0.1,

el método de Newton consigue perfectamente estabilizarse, sin embargo, el gradiente descente no consigue estabilizarse en un mínimo debido al salto demasiado grande entre sus iteraciones. Si quisiesemos que se estabilizase el gradiente descendente tendríamos que reducir su tasa de aprendizaje, siendo mucho más lento su avance como se comentaba en el ejercicio 1 y siendo más lento que el método de Newton. Cuidado, porque el que el método de Newton se consiga estabilizar no quiere decir que esté encontrando el mínimo. Esto nos lleva a la segunda conclusión.

Como segunda conclusión vemos que en el método de Newton se está dependiendo encontrar el mínimo del punto de inicio. En los puntos (1,1) y (3,3) vemos cómo la función llanea en el 2.9, antes de encontrar el mínimo. Esto se debe a que el método de Newton depende mucho de alcanzar el mínimo de si el punto de inicio está cerca de este mínimo.

Como conclusión final diremos que si queremos utilizar el método de Newton por ser más rápido estaremos dependiendo el punto de inicio y deberíamos lanzar el algoritmo con diferentes puntos de inicio. Si por otra parte queremos evitar tener que lanzar el algoritmo para varios puntos, podemos utilizar el gradiente descendente con una tasa pequeña; será más lento pero podremos estabilizarnos en torno a un mínimo.