Data Analysis and Machine Learning Lectures: Optimization and Gradient Methods

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Sep 27, 2018

Optimization, the central part of any Machine Learning algortithm

Almost every problem in machine learning and data science starts with a dataset X, a model $g(\beta)$, which is a function of the parameters β and a cost function $C(X, g(\beta))$ that allows us to judge how well the model $g(\beta)$ explains the observations X. The model is fit by finding the values of β that minimize the cost function. Ideally we would be able to solve for β analytically, however this is not possible in general and we must use some approximative/numerical method to compute the minimum.

Revisiting our Logistic Regression case

In our discussion on Logistic Regression we defined we studied first the case of two classes, with y_i either 0 or 1. Furthermore we assumed also that we have only two parameters β in our fitting of the Sigmoid function, that is we defined probabilities

$$p(y_i = 1|x_i, \hat{\beta}) = \frac{\exp{(\beta_0 + \beta_1 x_i)}}{1 + \exp{(\beta_0 + \beta_1 x_i)}},$$
$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}),$$

where $\hat{\beta}$ are the weights we wish to extract from data, in our case β_0 and β_1.

The equations to solve

Our compact equations used a definition of a vector $\hat{y}$ with n elements y_i, an $n \times p$ matrix $\hat{X}$ which contains the x_i values and a vector $\hat{p}$ of fitted probabilities $p(y_i|x_i, \hat{\beta})$. We rewrote in a more compact form the first derivative of the cost function as

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T \left(\hat{y} - \hat{p}\right).$$

If we in addition define a diagonal matrix $\hat{W}$ with elements $p(y_i|x_i, \hat{\beta})(1 - p(y_i|x_i, \hat{\beta})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2 \mathcal{C}(\hat{\beta})}{\partial \hat{\beta} \partial \hat{\beta}^T} = \hat{X}^T \hat{W} \hat{X}.$$

This defines what we call the Hessian.

Solving using Newton-Raphson's method

If we can set up these equations, Newton-Raphson's iterative method is the nomrally the method of choice. It requires however that we setting the matrices that define the first and second derivatives.

Our iterative scheme is then given by

$$\hat{\beta}^{\text{new}} = \hat{\beta}^{\text{old}} - \left(\frac{\partial^2 \mathcal{C}(\hat{\beta})}{\partial \hat{\beta} \partial \hat{\beta}^T}\right)^{-1} \times \left(\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}}\right)_{\hat{\beta}^{\text{old}}},$$

or in matrix form as

$$\hat{\beta}^{\text{new}} = \hat{\beta}^{\text{old}} - \left(\hat{X}^T \hat{W} \hat{X}\right)^{-1} \times \left(-\hat{X}^T (\hat{y} - \hat{p})\right)_{\hat{\beta}^{\text{old}}}.$$

The right-hand side is computed with the old values of β.

If we can compute these matrices, in particular the Hessian, the above is often the easiest method to implement.

Brief reminder on Newton-Raphson's method

Let us quicly remind ourselves how we derive the above method.

Perhaps the most celebrated of all one-dimensional root-finding routines is Newton's method, also called the Newton-Raphson method. This method is distinguished from the previously discussed methods by the fact that it requires the evaluation of both the function f and its derivative f' at arbitrary points. In this sense, it is taylored to cases with e.g., transcendental equations. If you can only calculate the derivative numerically and/or your function is not of the smooth type, we discourage the use of this method.

The equations

The Newton-Raphson formula consists geometrically of extending the tangent line at a current point until it crosses zero, then setting the next guess to the abscissa of that zero-crossing. The mathematics behind this method is rather simple. Employing a Taylor expansion for x sufficiently close to the solution s, we have

$$f(s) = 0 = f(x) + (s - x)f'(x) + \frac{(s - x)^2}{2} f''(x) + \dots .$$

For small enough values of the function and for well-behaved functions, the terms beyond linear are unimportant, hence we obtain

$$f(x) + (s - x)f'(x) \approx 0,$$

yielding

$$s \approx x - \frac{f(x)}{f'(x)}.$$

Having in mind an iterative procedure, it is natural to start iterating with

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

Simple geometric interpretation

The above is Newton-Raphson's method. It has a simple geometric interpretation, namely x_{n+1} is the point where the tangent from $(x_n, f(x_n))$ crosses the $x-$axis. Close to the solution, Newton-Raphson converges fast to the desired result. However, if we are far from a root, where the higher-order terms in the series are important, the Newton-Raphson formula can give grossly inaccurate results. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson may fail totally

Extending to more than one variable

Newton's method can be generalized to systems of several non-linear equations and variables. Consider the case with two equations

$$\begin{array}{ll} f_1(x_1, x_2) & = 0 \\ f_2(x_1, x_2) & = 0 \end{array},$$

which we Taylor expand to obtain

$$\begin{array}{lll} 0 = f_1(x_1 + h_1, x_2 + h_2) = & f_1(x_1, x_2) + h_1 \partial f_1/\partial x_1 + h_2 \partial f_1/\partial x_2 + \dots \\ 0 = f_2(x_1 + h_1, x_2 + h_2) = & f_2(x_1, x_2) + h_1 \partial f_2/\partial x_1 + h_2 \partial f_2/\partial x_2 + \dots \end{array}.$$

Defining the Jacobian matrix $\hat{\mathbf{J}}$ we have

$$\hat{\mathbf{J}} = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{pmatrix},$$

we can rephrase Newton's method as

$$\begin{pmatrix} x_1^{n+1} \\ x_2^{n+1} \end{pmatrix} = \begin{pmatrix} x_1^n \\ x_2^n \end{pmatrix} + \begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix},$$

where we have defined

$$\begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix} = -\hat{\mathbf{J}}^{-1} \begin{pmatrix} f_1(x_1^n, x_2^n) \\ f_2(x_1^n, x_2^n) \end{pmatrix}.$$

We need thus to compute the inverse of the Jacobian matrix and it is to understand that difficulties may arise in case $\hat{\mathbf{J}}$ is nearly singular.

It is rather straightforward to extend the above scheme to systems of more than two non-linear equations. In our case, the Jacobian matrix is given by the Hessian that represents the second derivative of cost function.

Steepest descent

The method of steepest descent The basic idea of gradient descent is that a function $F(\mathbf{x})$, $\mathbf{x} \equiv (x_1, \cdots, x_n)$, decreases fastest if one goes from $\mathbf{x}$ in the direction of the negative gradient $-\nabla F(\mathbf{x})$.

It can be shown that if

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k),$$

with $\gamma_k > 0$.

For γ_k small enough, then $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$. This means that for a sufficiently small γ_k we are always moving towards smaller function values, i.e a minimum.

More on Steepest descent

The previous observation is the basis of the method of steepest descent, which is also referred to as just gradient descent (GD). One starts with an initial guess $\mathbf{x}_0$ for a minimum of F and computes new approximations according to

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k), \quad k \geq 0.$$

The parameter γ_k is often referred to as the step length or the learning rate within the context of Machine Learning.

The ideal

Ideally the sequence $\{\mathbf{x}_k\}_{k=0}$ converges to a global minimum of the function F. In general we do not know if we are in a global or local minimum. In the special case when F is a convex function, all local minima are also global minima, so in this case gradient descent can converge to the global solution. The advantage of this scheme is that it is conceptually simple and straightforward to implement. However the method in this form has some severe limitations:

In machine learing we are often faced with non-convex high dimensional cost functions with many local minima. Since GD is deterministic we will get stuck in a local minimum, if the method converges, unless we have a very good intial guess. This also implies that the scheme is sensitive to the chosen initial condition.

Note that the gradient is a function of $\mathbf{x} = (x_1, \cdots, x_n)$ which makes it expensive to compute numerically.

The sensitiveness of the gradient descent

The gradient descent method is sensitive to the choice of learning rate γ_k. This is due to the fact that we are only guaranteed that $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$ for sufficiently small γ_k. The problem is to determine an optimal learning rate. If the learning rate is chosen too small the method will take a long time to converge and if it is too large we can experience erratic behavior.

Many of these shortcomings can be alleviated by introducing randomness. One such method is that of Stochastic Gradient Descent (SGD), see below.

Convex functions

Ideally we want our cost/loss function to be convex(concave).

First we give the definition of a convex set: A set C in $\mathbb{R}^n$ is said to be convex if, for all x and y in C and all $t \in (0, 1)$, the point $(1 - t)x + ty$ also belongs to C. Geometrically this means that every point on the line segment connecting x and y is in C as discussed below.

The convex subsets of $\mathbb{R}$ are the intervals of $\mathbb{R}$. Examples of convex sets of $\mathbb{R}^2$ are the regular polygons (triangles, rectangles, pentagons, etc...).

Convex function

Convex function: Let $X \subset \mathbb{R}^n$ be a convex set. Assume that the function $f : X \to \mathbb{R}$ is continuous, then f is said to be convex if

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

for all $x_1, x_2 \in X$ and for all $t \in [0, 1]$. If $\leq$ is replaced with a strict inequaltiy in the definition, we demand $x_1 \neq x_2$ and $t \in (0, 1)$ then f is said to be strictly convex. For a single variable function, convexity means that if you draw a straight line connecting $f(x_1)$ and $f(x_2)$, the value of the function on the interval $[x_1, x_2]$ is always below the line as illustrated below.

Conditions on convex functions

In the following we state first and second-order conditions which ensures convexity of a function f. We write D_f to denote the domain of f, i.e the subset of R^n where f is defined. For more details and proofs we refer to: S. Boyd and L. Vandenberghe. Convex Optimization. Cambridge University Press.

First order condition. Suppose f is differentiable (i.e $\nabla f(x)$ is well defined for all x in the domain of f). Then f is convex if and only if D_f is a convex set and

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

holds for all $x, y \in D_f$. This condition means that for a convex function the first order Taylor expansion (right hand side above) at any point a global under estimator of the function. To convince yourself you can make a drawing of $f(x) = x^2 + 1$ and draw the tangent line to $f(x)$ and note that it is always below the graph.

Second order condition. Assume that f is twice differentiable, i.e the Hessian matrix exists at each point in D_f. Then f is convex if and only if D_f is a convex set and its Hessian is positive semi-definite for all $x \in D_f$. For a single-variable function this reduces to $f''(x) \geq 0$. Geometrically this means that f has nonnegative curvature everywhere.

This condition is particularly useful since it gives us an procedure for determining if the function under consideration is convex, apart from using the definition.

More on convex functions

The next result is of great importance to us and the reason why we are going on about convex functions. In machine learning we frequently have to minimize a loss/cost function in order to find the best parameters for the model we are considering.

Ideally we want the global minimum (for high-dimensional models it is hard to know if we have local or global minimum). However, if the cost/loss function is convex the following result provides invaluable information:

Any minimum is global for convex functions. Consider the problem of finding $x \in \mathbb{R}^n$ such that $f(x)$ is minimal, where f is convex and differentiable. Then, any point x^* that satisfies $\nabla f(x^*) = 0$ is a global minimum.

This result means that if we know that the cost/loss function is convex and we are able to find a minimum, we are guaranteed that it is a global minimum.

Some simple problems

1. Show that $f(x) = x^2$ is convex for $x \in \mathbb{R}$ using the definition of convexity. Hint: If you re-write the definition, f is convex if the following holds for all $x, y \in D_f$ and any $\lambda \in [0, 1]$ $\lambda f(x) + (1 - \lambda)f(y) - f(\lambda x + (1 - \lambda)y) \geq 0$.

2. Using the second order condition show that the following functions are convex on the specified domain.

   - $f(x) = e^x$ is convex for $x \in \mathbb{R}$.
   - $g(x) = -\ln(x)$ is convex for $x \in (0, \infty)$.

3. Let $f(x) = x^2$ and $g(x) = e^x$. Show that $f(g(x))$ and $g(f(x))$ is convex for $x \in \mathbb{R}$. Also show that if $f(x)$ is any convex function than $h(x) = e^{f(x)}$ is convex.

4. A norm is any function that satisfy the following properties

   - $f(\alpha x) = |\alpha| f(x)$ for all $\alpha \in \mathbb{R}$.
   - $f(x + y) \leq f(x) + f(y)$
   - $f(x) \leq 0$ for all $x \in \mathbb{R}^n$ with equality if and only if $x = 0$

Using the definition of convexity, try to show that a function satisfying the properties above is convex (the third condition is not needed to show this).

Standard steepest descent

Before we proceed, we would like to mention the approach called the **standard Steepest descent**, which again leads to us having to be able to compute a matrix.

The success of the CG method for finding solutions of non-linear problems is based on the theory of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\hat{A}\hat{x} = \hat{b}.$$

In the iterative process we end up with a problem like

$$\hat{r} = \hat{b} - \hat{A}\hat{x},$$

where $\hat{r}$ is the so-called residual or error in the iterative process.

When we have found the exact solution, $\hat{r} = 0$.

Gradient method

The residual is zero when we reach the minimum of the quadratic equation

$$P(\hat{x}) = \frac{1}{2}\hat{x}^T \hat{A}\hat{x} - \hat{x}^T \hat{b},$$

with the constraint that the matrix $\hat{A}$ is positive definite and symmetric. If we search for a minimum of the quantum mechanical variance, then the matrix $\hat{A}$, which is called the Hessian, is given by the second-derivative of the function we want to minimize. This quantity is always positive definite.

Steepest descent method

We denote the initial guess for $\hat{x}$ as $\hat{x}_0$. We can assume without loss of generality that

$$\hat{x}_0 = 0,$$

or consider the system

$$\hat{A}\hat{z} = \hat{b} - \hat{A}\hat{x}_0,$$

instead.

Steepest descent method

One can show that the solution $\hat{x}$ is also the unique minimizer of the quadratic form

$$f(\hat{x}) = \frac{1}{2}\hat{x}^T \hat{A}\hat{x} - \hat{x}^T \hat{x}, \quad \hat{x} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{p}_1$ to be the gradient of f at $\hat{x} = \hat{x}_0$, which equals

$$\hat{A}\hat{x}_0 - \hat{b},$$

and $\hat{x}_0 = 0$ it is equal $-\hat{b}$.

Gradient descent method

Let $\hat{r}_k$ be the residual at the k-th step:

$$\hat{r}_k = \hat{b} - \hat{A}\hat{x}_k.$$

Note that $\hat{r}_k$ is the negative gradient of f at $\hat{x} = \hat{x}_k$, so the gradient descent method would be to move in the direction $\hat{r}_k$. This gives the following expression

$$\hat{p}_{k+1} = \hat{r}_k - \frac{\hat{p}_k^T \hat{A}\hat{r}_k}{\hat{p}_k^T \hat{A}\hat{p}_k}\hat{p}_k.$$

Final expressions

We can also compute the residual iteratively as

$$\hat{r}_{k+1} = \hat{b} - \hat{A}\hat{x}_{k+1},$$

which equals

$$\hat{b} - \hat{A}(\hat{x}_k + \alpha_k \hat{p}_k),$$

or

$$(\hat{b} - \hat{A}\hat{x}_k) - \alpha_k \hat{A}\hat{p}_k,$$

which gives

$$\hat{r}_{k+1} = \hat{r}_k - \hat{A}\hat{p}_k,$$

The Steepest descent algorithm

Simple codes for steepest descent and conjugate gradient using a 2×2 matrix, in c++, Python code to come

```
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "vectormatrixclass.h"
using namespace  std;
//   Main function begins here
int main(int  argc, char * argv[]){
  int dim = 2;
  Vector x(dim),xsd(dim), b(dim),x0(dim);
  Matrix A(dim,dim);

  // Set our initial guess
  x0(0) = x0(1) = 0;
  // Set the matrix
  A(0,0) =  3;    A(1,0) =  2;   A(0,1) =  2;   A(1,1) =  6;
  b(0) = 2; b(1) = -8;
  cout << "The Matrix A that we are using: " << endl;
  A.Print();
  cout << endl;
  xsd = SteepestDescent(A,b,x0);
  cout << "The approximate solution using Steepest Descent is: " << endl;
  xsd.Print();
  cout << endl;
}
```

The routine for the steepest descent method

```
Vector SteepestDescent(Matrix A, Vector b, Vector x0){
  int IterMax, i;
  int dim = x0.Dimension();
```

```
  const double tolerance = 1.0e-14;
  Vector x(dim),f(dim),z(dim);
  double c,alpha,d;
  IterMax = 30;
  x = x0;
  f = A*x-b;
  i = 0;
  while (i <= IterMax){
    z = A*f;
    c = dot(f,f);
    alpha = c/dot(f,z);
    x = x - alpha*f;
    f =  A*x-b;
    if(sqrt(dot(f,f)) < tolerance) break;
    i++;
  }
  return x;
}
```

Revisiting our first homework

We will use linear regression as a case study for the gradient descent methods. Linear regression is a great test case for the gradient descent methods discussed in the lectures since it has several desirable properties such as:

1. An analytical solution (recall homework set 1).
2. The gradient can be computed analytically.
3. The cost function is convex which guarantees that gradient descent converges for small enough learning rates

We revisit the example from homework set 1 where we had

$$y_i = 5x_i^2 + 0.1\xi_i, \ i = 1, \cdots, 100$$

with $x_i \in [0, 1]$ chosen randomly with a uniform distribution. Additionally ξ_i represents stochastic noise chosen according to a normal distribution $\mathcal{N}(\prime, \infty)$. The linear regression model is given by

$$h_\beta(x) = \hat{y} = \beta_0 + \beta_1 x,$$

such that

$$\hat{y}_i = \beta_0 + \beta_1 x_i.$$

Gradient descent example

Let $\mathbf{y} = (y_1, \cdots, y_n)^T$, $\hat{\mathbf{y}} = (\hat{y}_1, \cdots, \hat{y}_n)^T$ and $\beta = (\beta_0, \beta_1)^T$

It is convenient to write $\hat{\mathbf{y}} = X\beta$ where $X \in \mathbb{R}^{100\times 2}$ is the design matrix given by

$$X \equiv \begin{bmatrix} 1 & amp; x_1 & \\ \vdots & amp; \vdots & \\ 1 & amp; x_{100} & amp; \end{bmatrix}.$$

The loss function is given by

$$C(\beta) = ||X\beta - \mathbf{y}||^2 = ||X\beta||^2 - 2\mathbf{y}^T X\beta + ||\mathbf{y}||^2 = \sum_{i=1}^{100} (\beta_0 + \beta_1 x_i)^2 - 2y_i(\beta_0 + \beta_1 x_i) + y_i^2$$

and we want to find β such that $C(\beta)$ is minimized.

The derivative of the cost/loss function

Computing $\partial C(\beta)/\partial\beta_0$ and $\partial C(\beta)/\partial\beta_1$ we can show that the gradient can be written as

$$\nabla_\beta C(\beta) = (\partial C(\beta)/\partial\beta_0, \partial C(\beta)/\partial\beta_1)^T = 2\begin{bmatrix} \sum_{i=1}^{100} (\beta_0 + \beta_1 x_i - y_i) \\ \sum_{i=1}^{100} (x_i(\beta_0 + \beta_1 x_i) - y_i x_i) \end{bmatrix} = 2X^T(X\beta - \mathbf{y}),$$

where X is the design matrix defined above.

The Hessian matrix

The Hessian matrix of $C(\beta)$ is given by

$$\hat{H} \equiv \begin{bmatrix} \frac{\partial^2 C(\beta)}{\partial\beta_0^2} & amp; \frac{\partial^2 C(\beta)}{\partial\beta_0\partial\beta_1} \\ \frac{\partial^2 C(\beta)}{\partial\beta_0\partial\beta_1} & amp; \frac{\partial^2 C(\beta)}{\partial\beta_1^2} \quad amp; \end{bmatrix} = 2X^T X.$$

This result implies that $C(\beta)$ is a convex function since the matrix $X^T X$ always is positive semi-definite.

Simple program

We can now write a program that minimizes $C(\beta)$ using the gradient descent method with a constant learning rate γ according to

$$\beta_{k+1} = \beta_k - \gamma\nabla_\beta C(\beta_k), \ k = 0, 1, \cdots$$

We can use the expression we computed for the gradient and let use a β_0 be chosen randomly and let $\gamma = 0.001$. Stop iterating when $||\nabla_\beta C(\beta_k)|| \leq \epsilon = 10^{-8}$.

And finally we can compare our solution for β with the analytic result given by $\beta = (X^T X)^{-1} X^T \mathbf{y}$.

```
import numpy as np

"""
The following setup is just a suggestion, feel free to write it the way you like.
"""

#Setup problem described in the exercise
N  = 100 #Nr of datapoints
M  = 2 #Nr of features
x  = np.random.rand(N) #Uniformly generated x-values in [0,1]
```

```
y  = 5*x**2 + 0.1*np.random.randn(N)
X  = np.c_[np.ones(N),x] #Construct design matrix

#Compute beta according to normal equations to compare with GD solution
Xt_X_inv = np.linalg.inv(np.dot(X.T,X))
Xt_y     = np.dot(X.transpose(),y)
beta_NE = np.dot(Xt_X_inv,Xt_y)
print(beta_NE)
```

Gradient Descent Example

Another simple example is here

```
# Importing various packages
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys

x = 2*np.random.rand(100,1)
y = 4+3*x+np.random.randn(100,1)

xb = np.c_[np.ones((100,1)), x]
beta_linreg = np.linalg.inv(xb.T.dot(xb)).dot(xb.T).dot(y)
print(beta_linreg)
beta = np.random.randn(2,1)

eta = 0.1
Niterations = 1000
m = 100

for iter in range(Niterations):
    gradients = 2.0/m*xb.T.dot(xb.dot(beta)-y)
    beta -= eta*gradients

print(beta)
xnew = np.array([[0],[2]])
xbnew = np.c_[np.ones((2,1)), xnew]
ypredict = xbnew.dot(beta)
ypredict2 = xbnew.dot(beta_linreg)
plt.plot(xnew, ypredict, "r-")
plt.plot(xnew, ypredict2, "b-")
plt.plot(x, y ,'ro')
plt.axis([0,2.0,0, 15.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Gradient descent example')
plt.show()
```

And a corresponding example using scikit-learn

```
# Importing various packages
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
```

```
x = 2*np.random.rand(100,1)
y = 4+3*x+np.random.randn(100,1)

xb = np.c_[np.ones((100,1)), x]
beta_linreg = np.linalg.inv(xb.T.dot(xb)).dot(xb.T).dot(y)
print(beta_linreg)
sgdreg = SGDRegressor(n_iter = 50, penalty=None, eta0=0.1)
sgdreg.fit(x,y.ravel())
print(sgdreg.intercept_, sgdreg.coef_)
```

Gradient descent and Ridge

We have also discussed Ridge regression where the loss function contains a regularized given by the L_2 norm of β,

$$C_{\text{ridge}}(\beta) = ||X\beta - \mathbf{y}||^2 + \lambda||\beta||^2, \ \lambda \geq 0.$$

In order to minimize $C_{\text{ridge}}(\beta)$ using GD we only have adjust the gradient as follows

$$\nabla_\beta C_{\text{ridge}}(\beta) = 2\begin{bmatrix} \sum_{i=1}^{100}(\beta_0 + \beta_1 x_i - y_i) \\ \sum_{i=1}^{100}(x_i(\beta_0 + \beta_1 x_i) - y_i x_i) \end{bmatrix} + 2\lambda\begin{bmatrix}\beta_0 \\ \beta_1\end{bmatrix} = 2(X^T(X\beta - \mathbf{y}) + \lambda\beta).$$

We can now extend our program to minimize $C_{\text{ridge}}(\beta)$ using gradient descent and compare with the analytical solution given by

$$\beta_{\text{ridge}} = \left(X^T X + \lambda I_{2\times 2}\right)^{-1} X^T \mathbf{y},$$

for $\lambda = 0, 1, 10, 50, 100$ ($\lambda = 0$ corresponds to ordinary least squares). We can then compute $||\beta_{\text{ridge}}||$ for each λ.

```
import numpy as np

"""
The following setup is just a suggestion, feel free to write it the way you like.
"""

#Setup problem described in the exercise
N  = 100 #Nr of datapoints
M  = 2   #Nr of features
x  = np.random.rand(N)
y  = 5*x**2 + 0.1*np.random.randn(N)


#Compute analytic beta for Ridge regression
X    = np.c_[np.ones(N),x]
XT_X = np.dot(X.T,X)

l  = 0.1 #Ridge parameter lambda
Id = np.eye(XT_X.shape[0])

Z = np.linalg.inv(XT_X+l*Id)
beta_ridge = np.dot(Z,np.dot(X.T,y))

print(beta_ridge)
print(np.linalg.norm(beta_ridge)) #||beta||
```

Stochastic Gradient Descent

Stochastic gradient descent (SGD) and variants thereof address some of the shortcomings of the Gradient descent method discussed above.

The underlying idea of SGD comes from the observation that the cost function, which we want to minimize, can almost always be written as a sum over n data points $\{\mathbf{x}_i\}_{i=1}^n$,

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta).$$

Computation of gradients

This in turn means that the gradient can be computed as a sum over i-gradients

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta).$$

Stochasticity/randomness is introduced by only taking the gradient on a subset of the data called minibatches. If there are n data points and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$.

SGD example

As an example, suppose we have 10 data points $(\mathbf{x}_1, \cdots, \mathbf{x}_{10})$ and we choose to have $M = 5$ minibathces, then each minibatch contains two data points. In particular we have $B_1 = (\mathbf{x}_1, \mathbf{x}_2), \cdots, B_5 = (\mathbf{x}_9, \mathbf{x}_{10})$. Note that if you choose $M = 1$ you have only a single batch with all data points and on the other extreme, you may choose $M = n$ resulting in a minibatch for each datapoint, i.e $B_k = \mathbf{x}_k$.

The idea is now to approximate the gradient by replacing the sum over all data points with a sum over the data points in one the minibatches picked at random in each gradient descent step

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \rightarrow \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta).$$

The gradient step

Thus a gradient descent step now looks like

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta)$$

where k is picked at random with equal probability from $[1, n/M]$. An iteration over the number of minibathces (n/M) is commonly referred to as an epoch. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches, as exemplified in the code below.

Simple example code

```
import numpy as np

n = 100 #100 datapoints
M = 5   #size of each minibatch
m = int(n/M) #number of minibatches
n_epochs = 10 #number of epochs

j = 0
for epoch in range(1,n_epochs+1):
    for i in range(m):
        k = np.random.randint(m) #Pick the k-th minibatch at random
        #Compute the gradient using the data in minibatch Bk
        #Compute new suggestion for
        j += 1
```

Taking the gradient only on a subset of the data has two important benefits. First, it introduces randomness which decreases the chance that our opmization scheme gets stuck in a local minima. Second, if the size of the minibatches are small relative to the number of datapoints ($M < n$), the computation of the gradient is much cheaper since we sum over the datapoints in the $k-th$ minibatch and not all n datapoints.

When do we stop?

A natural question is when do we stop the search for a new minimum? One possibility is to compute the full gradient after a given number of epochs and check if the norm of the gradient is smaller than some threshold and stop if true. However, the condition that the gradient is zero is valid also for local minima, so this would only tell us that we are close to a local/global minimum. However, we could also evaluate the cost function at this point, store the result and continue the search. If the test kicks in at a later stage we can compare the values of the cost function and keep the β that gave the lowest value.

Slightly different approach

Another approach is to let the step length γ_j depend on the number of epochs in such a way that it becomes very small after a reasonable time such that we do not move at all.

As an example, let $e = 0, 1, 2, 3, \cdots$ denote the current epoch and let $t_0, t_1 > 0$ be two fixed numbers. Furthermore, let $t = e \cdot m + i$ where m is the number of minibatches and $i = 0, \cdots, m-1$. Then the function

$$\gamma_j(t; t_0, t_1) = \frac{t_0}{t + t_1}$$

goes to zero as the number of epochs gets large. I.e. we start with a step length $\gamma_j(0; t_0, t_1) = t_0/t_1$ which decays in *time* t.

In this way we can fix the number of epochs, compute β and evaluate the cost function at the end. Repeating the computation will give a different result

since the scheme is random by design. Then we pick the final β that gives the lowest value of the cost function.

```
import numpy as np

def step_length(t,t0,t1):
    return t0/(t+t1)

n = 100 #100 datapoints
M = 5   #size of each minibatch
m = int(n/M) #number of minibatches
n_epochs = 500 #number of epochs
t0 = 1.0
t1 = 10

gamma_j = t0/t1
j = 0
for epoch in range(1,n_epochs+1):
    for i in range(m):
        k = np.random.randint(m) #Pick the k-th minibatch at random
        #Compute the gradient using the data in minibatch Bk
        #Compute new suggestion for beta
        t = epoch*m+i
        gamma_j = step_length(t,t0,t1)
        j += 1

print("gamma_j after %d epochs: %g" % (n_epochs,gamma_j))
```