ORT BRAUDE COLLEGE

DEPARTMENT OF MATHEMATICS
B.SC. FINAL YEAR PROJECT

First order gradient methods

Author:
Shira Bar-Dov

Supervisor:
Assoc. Prof. Aviv Gibali

July 19, 2019

המכללה האקדמית להנדסה
אורט בראודה
ORT BRAUDE COLLEGE

Contents

1 Abstract

In this project we describe and analyze different kinds of iterative first order gradient algorithms for solving optimization problems arising in image processing and support vector machines.
This class of methods, which can be viewed as an extension of the classical gradient algorithms, are attractive due to their simplicity and thus they adequate for solving large-scale problems even with dense matrix data. However, such methods are also known to converge quite slowly.
In this project we present several powerful iterative descent method for finding a local minimum of a multivariable function, that have been proven to converge faster than classical gradient algorithms.
Numerical experiments for image deblurring and classification problems are given.

2 Introduction

In this project we are concerned with optimization models for data fitting and data classification. Specifically, we focus on first order gradient descent methods such as ISTA- Iterative Shrinkage Thresholding Algorithm and FISTA- Fast Iterative Shrinkage Thresholding Algorithm for data fitting and a stochastic method called Pegasos for data classification.
Gradient descent was invented by the french mathematician Louis Augustin Cauchy in 1847. Since then, gradient descent methods kept developing, subgradients methods and stochastic subgradient methods from convex optimization were discovered during 1960-1970.
Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. It is a common optimization method in machine learning algorithms, which are based on a convex function. Through an iterative process and the use of partial differential equations, gradient descent constructs a set of parameters, in order to minimize a given cost function to its local minimum.
Gradient descent is based on the observation that if the multivariable function $F(x)$ is defined and differentiable in a neighborhood of a point x , then $F(x)$ decreases fastest if one goes from x in the direction of the negative gradient of F at x ,$-\nabla F(x)$. It follows that, if

$$x_{n+1} = x_n - \gamma \nabla F(x_n).$$

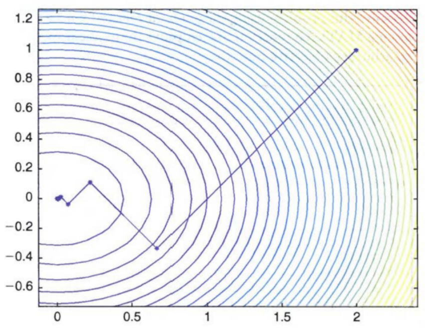


Figure 1: The iterates of the gradient method along with the contour lines of the objective function

for $\gamma \in \mathbb{R}_+$ small enough, the term $\gamma \nabla F(x)$ is subtracted from x because we want to move against the gradient, toward the minimum.

The stepsize or the learning rate γ is a positive number that determines the size of each step in the process. If it is too small, the process could be slow. On the other hand, if it is too large the process can skip the minimum and may not converge.
With this in mind, consider the unconstrained nonsmooth convex optimization problem

$$\min_x \{F(x) \equiv f(x) + g(x) : x \in \mathbb{R}^n\}, \tag{1}$$

where f, g are convex function, with g possibly nonsmooth.
The gradient algorithm is one of the simplest methods for solving (1). It generates a sequence $\{x_k\}$ via

$$x_0 \in \mathbb{R}^n, \quad x_k = x_{k-1} - t_k(\nabla f(x_{k-1}) + \nabla g(x_{k-1})),$$

where $t_k > 0$ is a suitable stepsize.

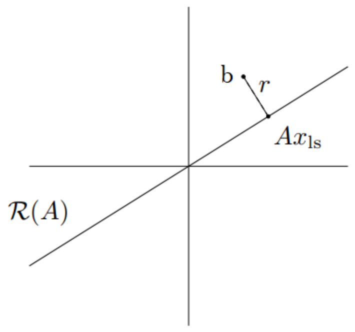
This problem arises at many different applications among them signal and image processing, optics, speech tagging and music identification.
When solving large-scale problems, first-order methods are often the only practical option, but the sequence x_k converges quite slowly to a solution.

In this project, we focus on two specific models, one for data fitting with and without regulators, and one for classification.
Data fitting and classifying data are common tasks in machine learning. If a parametrized model function meant to explain some phenomena is given, the goal is to adjust the numerical values for the model so that it most closely matches some data.

In data fitting problems, a classical approach to the familiar basic linear problem (1) where

$$f(x) = Ax - b, \; g(x) = 0,$$

where $A \in \mathbb{R}^{m \times n}, \; x \in \mathbb{R}^n, \; b \in \mathbb{R}^m, \; R(A)$ *the column space of* A.
is the least squares, which estimate the solution by minimize the data error. In other words, if we define the error $r = Ax - b$ then we find $x = x_{ls}$ that minimize $\|r\|$. x_{ls} is called least-squares approximate solution of $Ax = b$. Ax_{ls} is point in $R(A)$ closest to b, and the projection of b onto $R(A)$.



Note that if $b \in R(A)$ then x_{ls} solves $Ax_{ls} = b$, or if A is square it is invertible (nonsingular) so $x_{ls} = A^{-1}b$.

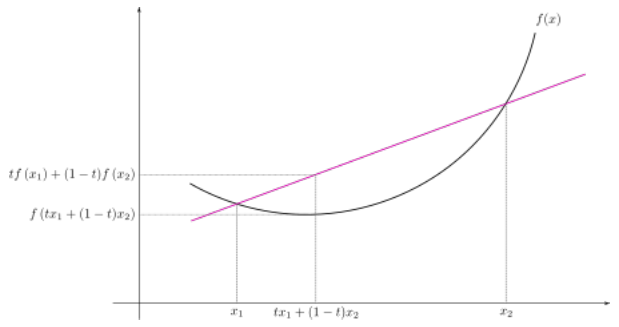
In classification problems, data points are given, and each belong to one of two or more classes. When a new point is given, the goal is to determine in which class it will be. In support vector machines, each data point is a p dimensional vector, and the goal is to determine whether it could be separated with a $p-1$ dimensional hyperplane. Note that there are many hyperplanes that might classify the data, but the best hyperplane is the one that represents the largest separation (margin) between the two classes. Hence, the optimal hyperplane is the one that maximized the distance from it to the nearest data point on each side.

3 Preliminaries

Convex function: A function f is convex if for every u, v in the domain, and for every $\lambda \in [0,1]$ we have

$$f(\lambda u + (1-\lambda)v) \leq \lambda f(u) + (1-\lambda) f(v).$$

In general, the necessary condition for x to be a minimum for the function f is $\nabla f(x) = 0$. For convex functions, this is both necessary and sufficient.



Geometrically, the inequality in the definition means the graph of f between u and v is below the segment which joins the points $(u, f(u))$ and $(v, f(v))$.

l_1 **Norm:** l_1 Norm is the sum of the magnitudes of the vectors in a space. A vector norm defined for a vector $\vec{x} = (x_1, ..., x_n)$ as $\|\vec{x}\|_1 = \sum_{i=1}^{n} |x_i|$.

Dense matrix: A matrix in which most of the elements are nonzero.

Lipschitz continuous gradient: A function f that uphold

$$\|\nabla f(x) - \nabla f(y)\| \leq L(f)\|x - y\| \quad for\ every\ x, y \in \mathbb{R}^n,$$

where $\|\cdot\|$ denotes the standard Euclidean norm and $L(f) > 0$ is the Lipschitz constant of ∇f.

Hyperplane: A hyperplane in an n dimensional Euclidean space, is a flat $n-1$ dimensional subset of that space that divides the space into two disconnected parts.

Inner product: The inner product of two vectors $\vec{a} = (a_1, a_2, ..., a_n)$ and $\vec{b} = (b_1, b_2, ..., b_n)$ is defined as:

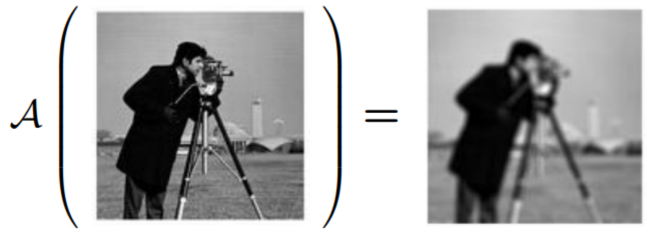
$$a \cdot b = a^T b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_n b_n.$$

Gradient: The gradient of a scalar multivariable function $f(x_1, x_2, ..., x_n)$ is denoted ∇f, packages all its partial derivative information into a vector: $\nabla f(\vec{x}) = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n})$.

Smooth function: A smooth function is a function that has derivatives of all orders everywhere in its domain.

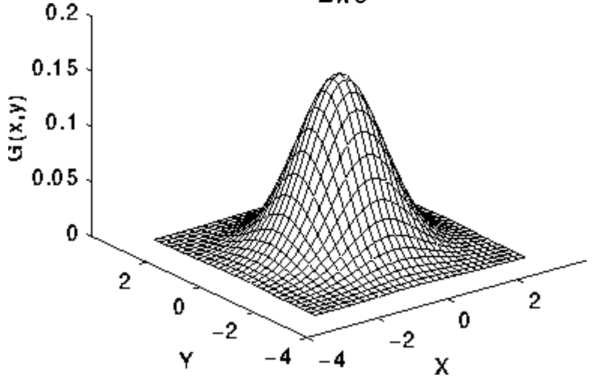
4 Data Fitting

In this section we discuss first order methods of gradient descent iterative algorithms, for solving linear inverse problems arising in image processing.
A basic linear inverse "image deblurring" problem $Ax = b$



where $A \in \mathbb{R}^{m \times n}$ represents a two-dimensional convolution blur operator, and $b \in \mathbb{R}^m$ represents the blurred image, are known. $x \in \mathbb{R}^n$ is the "true" and unknown image to be estimated, its size is assumed to be the same as that of b ($n = m$). Both x and b are formed by stacking the columns of their corresponding two-dimensional images.
The blur operator or the Gaussian operator is a two-dimensional convolution operator that is used to blur images and remove detail and noise. The idea of Gaussian smoothing is to use this two-dimensional distribution as a 'point-spread' function, and this is achieved by convolution of a Gaussian mask h and the true image x.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



4.1 ISTA

The class of Iterative Shrinkage Thresholding Algorithms (ISTA) is an extension of the classical gradient algorithm. It is attractive due to its simplicity and thus is adequate for solving large-scale problems even with dense matrix data.

Adopting this same gradient idea to the problem formulation (1) where the following assumptions are made:
$g : \mathbb{R}^n \to \mathbb{R}$ is a continuous convex function which is possibly nonsmooth.
$f : \mathbb{R}^n \to \mathbb{R}$ is a continuously differentiable with Lipschitz continuous gradient $L(f)$.

Note that when $g(x) \equiv 0$, (1) is the general unconstrained smooth convex minimization problem.
When $f(x) = \|Ax - b\|^2$, $g(x) \equiv \|x\|_1$, (1) is the $l1$ regularization problem

$$min\left\{f(x) + \lambda\|x\|_1 : x \in \mathbb{R}^n\right\}.$$

To calculate the Lipschitz constant of the gradient ∇f we first write f as follows:

$$f(x) = \|Ax - b\|^2 = (Ax - b)^T(Ax - b) = x^TA^TAx - x^TA^Tb - b^TAx + b^Tb =$$
$$= x^TA^TAx - (b^TAx)^T - b^TAx + b^Tb = x^TA^TAx - 2b^TAx + b^Tb.$$

The last transaction is because the product b^TAx is a scalar product of vector b and Ax, and therefore $(b^TAx)^T = b^TAx$.
The gradient of $\nabla f(x)$:

$$\nabla f(x) = \nabla(x^TA^TAx - 2b^TAx + b^Tb) = (A^TA + (A^TA)^T)x - 2(b^TA)^T = 2A^TAx - 2A^Tb.$$

We define the Lipschitz continuous gradient at the preliminaries section,

$$\|\nabla f(x) - \nabla f(y)\| = \|2A^TAx - 2A^Tb - (2A^TAy - 2A^Tb)\| = \|2A^TA(x - y)\| \leq 2\|A^TA\|\|x - y\|.$$

The inequality is a result of the Cauchy Bunyakovsky Schwarz inequality, therefore the Lipschitz constant of the gradient is $2\|A^TA\|$.

Theorem 1 *Let A be a symmetric $n \times n$ matrix, $\|A\| = max_j|\lambda_j|$, where λ_j are the eigenvalues of A.*

Proof. The norm of a matrix is defined as

$$\|A\| = \max_{\|u\|=1} \|Au\|.$$

Taking the singular value decomposition of the matrix A, we have

$$A = VDW^T,$$

where V and W are re orthonormal and D is a diagonal matrix. Since V and W are orthonormal, we have $\|V\| = 1$ and $\|W\| = 1$. Then $\|Av\| = \|Dv\|$ for any vector v. Then we can maximize the norm of Av by maximizing the norm of Dv. By the definition of singular value decomposition, D will have the singular values of A on its main diagonal and will have zeros everywhere else. Let $\lambda_1, ..., \lambda_n$ denote these diagonal entries so that

$$D = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}.$$

Taking some $v = (v_1, v_2, \cdots, v_n)^T$, the product Dv takes the form

$$Dv = \begin{pmatrix} \lambda_1 v_1 \\ \vdots \\ \lambda_n v_n \end{pmatrix}.$$

Maximizing the norm of this is the same as maximizing the norm squared. Then we are trying to maximize the sum

$$S = \sum_{i=1}^{n} \lambda_i^2 v_i^2,$$

under the constraint that v is a unit vector (i.e., $\sum_i v_i^2 = 1$). The maximum is attained by finding the largest λ_i^2 and setting its corresponding v_i to 1 and then setting each other v_j to 0. Then the maximum of S (which is the norm squared) is the square of the absolutely largest eigenvalue of A. Taking the square root, we get the absolutely largest eigenvalue of A. ■

A^TA is a symmetric matrix, hence, the smallest Lipschitz constant of the gradient ∇f is $L(f) = 2\|A^TA\| = 2\lambda_{max}(A^TA)$, where λ_{max} is the maximal eigenvalue of A^TA.

One of the methods for solving (1) is in the class of ISTA. The ISTA method which generates a sequence $\{x_k\}$ in this case would be:

$$x_0 \in \mathbb{R}^n, \quad x_k = x_{k-1} - \frac{1}{L}(\nabla f(x_{k-1}) + \nabla g(x_{k-1})). \tag{2}$$

The ISTA iteration (2) can be viewed as a proximal regularization of the linearized function f at x_{k-1}, and written equivalently as

$$x_k = \underset{x}{\operatorname{argmin}}\{f(x_{k-1}) + \langle x - x_{k-1}, \nabla f(x_{k-1})\rangle + \frac{L}{2}\|x - x_{k-1}\|^2 + g(x)\}.$$

For any $L > 0$, consider the quadratic approximation of (1) at a given point y:

$$Q_L(x, y) := f(y) + \langle x - y, \nabla f(y)\rangle + \frac{L}{2}\|x - y\|^2 + g(x),$$

which admits a unique minimizer

$$p_L(y) := \operatorname{argmin}\left\{Q_L(x, y) : x \in \mathbb{R}^n\right\}.$$

Simple Algebra shows that

$$Q_L(x, y) := f(y) + \langle x - y, \nabla f(y)\rangle + \frac{L}{2}\langle x - y, x - y\rangle + g(x) =$$

$$= g(x) + \frac{L}{2}\langle x - y, \frac{2}{L}\nabla f(y)\rangle + \frac{L}{2}\langle x - y, x - y\rangle + f(y) =$$

$$= g(x) + \frac{L}{2}\left(\langle x - y, x - y\rangle + 2\langle x - y, \frac{1}{L}\nabla f(y)\rangle\right) + f(y) =$$

$$= g(x) + \frac{L}{2}\left(\langle x - y + \frac{1}{L}\nabla f(y), x - y + \frac{1}{L}\nabla f(y)\rangle - \langle \frac{1}{L}\nabla f(y), \frac{1}{L}\nabla f(y)\rangle\right) + f(y) =$$

$$= g(x) + \frac{L}{2}\left\|x - y + \frac{1}{L}\nabla f(y)\right\|^2 - \frac{1}{2L}\|\nabla f(y)\| + f(y).$$

ignoring constant terms in y

$$p_L(y) = \underset{x}{\operatorname{argmin}} \left\{ g(x) + \frac{L}{2} \left\| x - \left(y - \frac{1}{L} \nabla f(y) \right) \right\|^2 \right\}.$$

Then, the gradient algorithm ISTA generates a sequence $\{x_k\}$ via

$$x_0 \in \mathbb{R}^n, \; x_k = p_L(x_{k-1}),$$

where $L > 0$ plays the role of a stepsize.

To conclude, the basic iteration of ISTA for solving problem (1) with constant stepsize is:

Input: $L := L(f)$- A Lipschitz constant of ∇f.
Initialize: Take $x_0 \in \mathbb{R}^n$
Step k: For each $k \geq 1$ compute

$$x_k = p_L(x_{k-1}).$$

4.2 FISTA

A Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) is an improvement of the ISTA method for solving the general problem (1). While FISTA keeps the simplicity of ISTA, the sequence x_k that FISTA generates converges more quickly to a solution than other gradient methods.
The main difference between the ISTA method and the FISTA method is that the iterative shrinkage operator $p_L()$ is not employed on the previous point x_{k-1}, but rather at the point y_k, which uses a very specific linear combination of the previous two points x_{k1}, x_{k2}. Obviously, the main computational effort in both ISTA and FISTA remains the same, because the requested additional computation for FISTA in the steps (4) and (5) is clearly marginal.

The basic iteration of FISTA for solving problem (1) with constant stepsize is:

Input: $L := L(f)$- A Lipschitz constant of ∇f.
Initialize: Take $y_1 = x_0 \in \mathbb{R}^n,\ t_1 = 1$
Step k: For each $k \geq 1$ compute

$$x_k = p_L(x_{k-1}) \tag{3}$$

$$t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2} \tag{4}$$

$$y_{k+1} = x_k + \left(\frac{t_k - 1}{t_{k+1}}\right). \tag{5}$$

4.3 Numerical experiments

In this section we illustrate an image deblurring problem. We estimate an image $\boldsymbol{x}$ from blurred image $\boldsymbol{b}$ using the iterative gradient descent algorithms ISTA and FISTA. We compare ISTA to FISTA and show the difference between the performance of those methods.
Both methods were used with a constant stepsize rule and applied once when

$$f(x) = \|Ax - b\|^2, \quad g(x) = 0, \tag{6}$$

and once with the l_1 regulator when

$$f(x) = \|Ax - b\|^2, \quad g(x) = \lambda\|x\|_1. \tag{7}$$

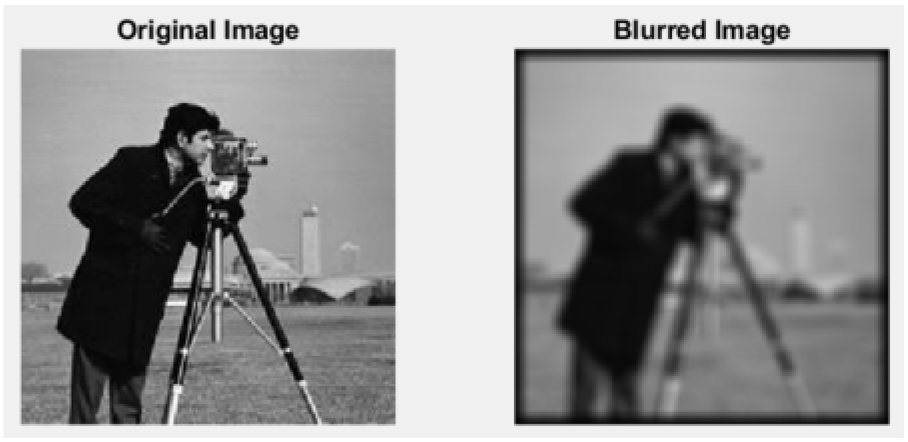


Figure 2: Deblurring of the cameraman

In this example we look at the 250×250 cameraman image. The image went through a Gaussian blur of size 9×9 and standard deviation 4 (applied by the MATLAB functions conv2 and fspecial). The original and blurred images are given in Figure (2).

We tested ISTA and FISTA for solving problems (6) and (7), where $\boldsymbol{b}$ represents the vectorized blurred image, $\boldsymbol{A}$ represents the blur operator, which is a two-dimensional convolution operator, and $\boldsymbol{x}$ represent the unknown true image. The regularization parameter has been chosen to be $\lambda = 1$. The Lipschitz constant was computable in this example, since the maximum eigenvalue of $A^T A$ can be calculated.

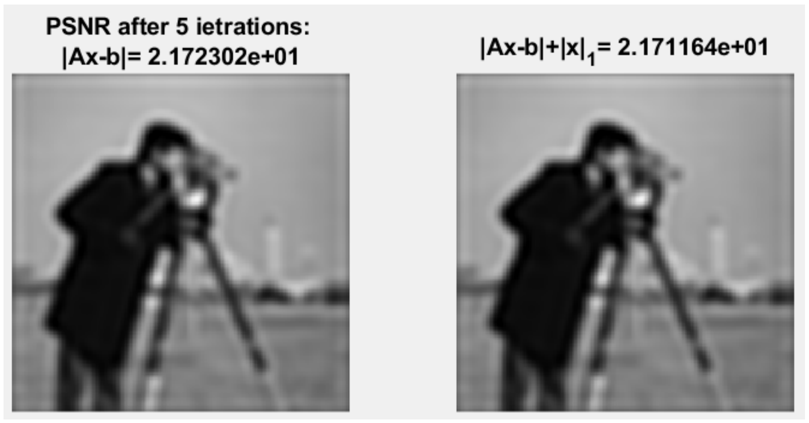
Iterations 5, 50, 100 and 200 of ISTA algorithm are described in Figure (3).

Iterations 5, 50, 100 and 200 of FISTA algorithm are described in Figure (4).

The peak signal to noise ratio (PSNR) for the reconstructed image is calculated, with the true image as the reference. This ratio denotes how close are we to the true image, where higher values would denote that the image is closer to the true image.

The function value that we want to minimize is given at those iterations, in order to track its value as it gets lower with the iterations.

ISTA's iterations:



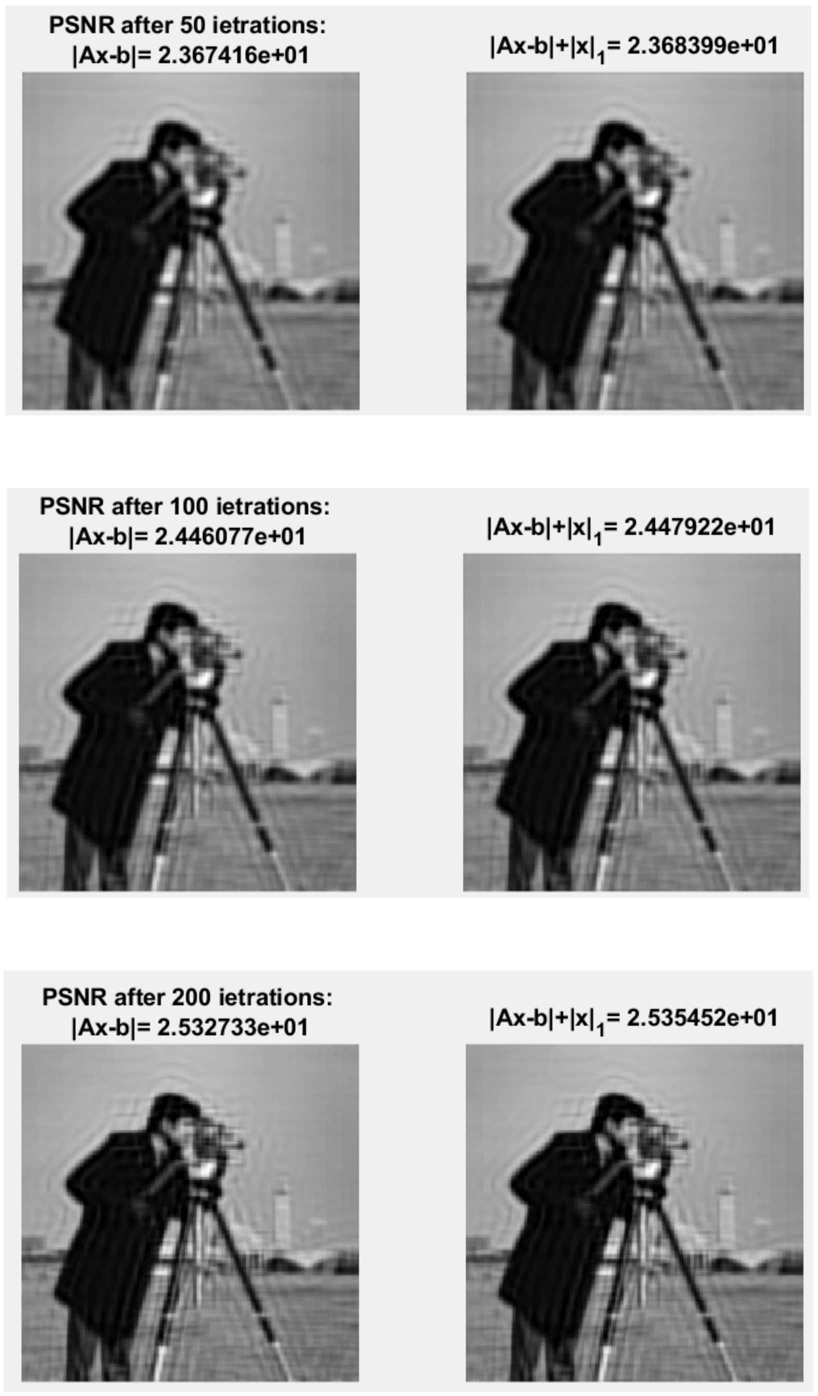
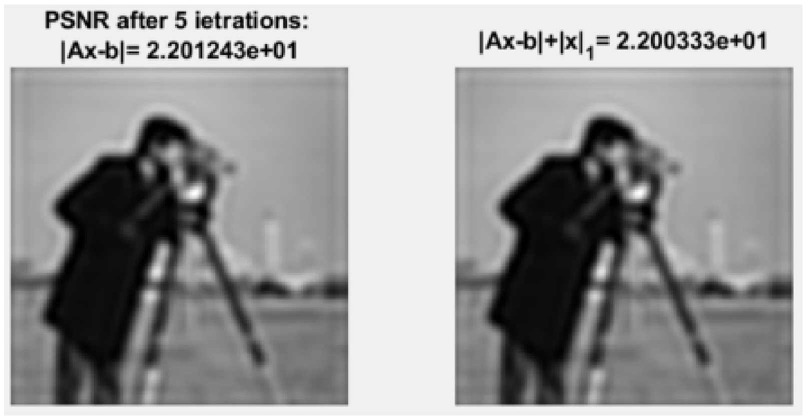
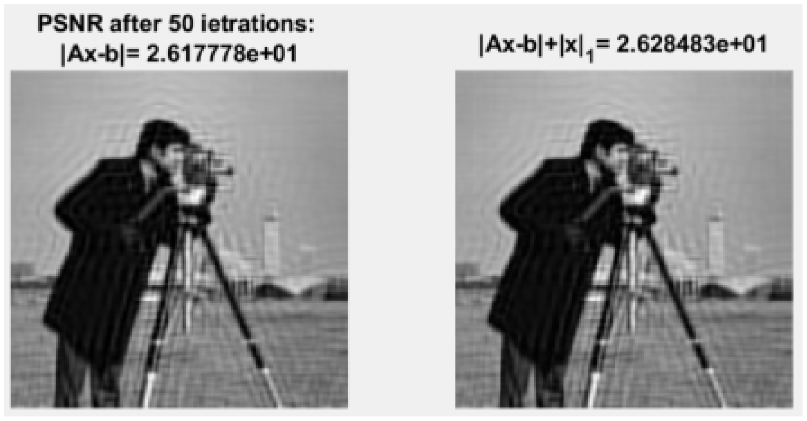
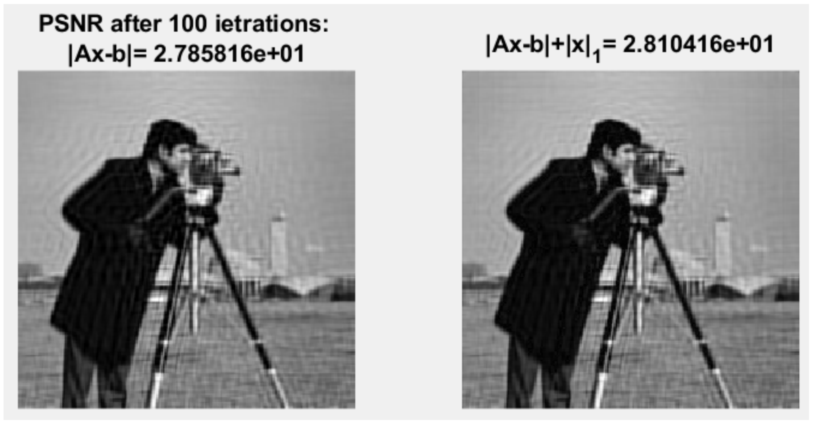

PSNR after 50 ietrations:
|Ax-b|= 2.367416e+01
|Ax-b|+|x|1= 2.368399e+01
PSNR after 100 ietrations:
|Ax-b|= 2.446077e+01
|Ax-b|+|x|1= 2.447922e+01
PSNR after 200 ietrations:
|Ax-b|= 2.532733e+01
|Ax-b|+|x|1= 2.535452e+01

Figure 3: Iterations of ISTA for deblurring of the cameraman.

FISTA's iterations:







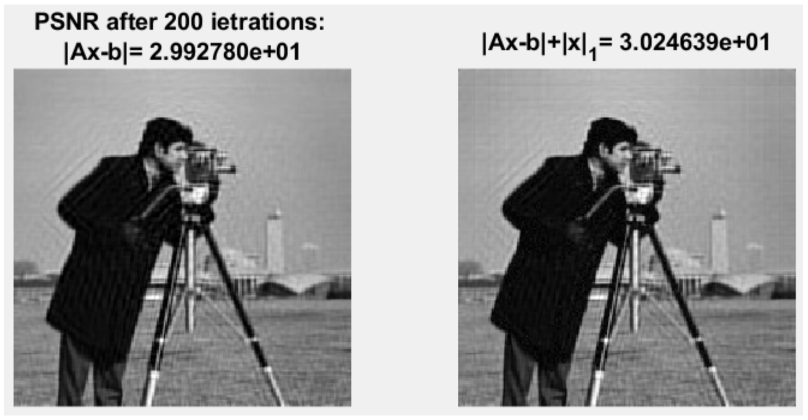


Figure 4: Iterations of FISTA for deblurring of the cameraman.

The data is summarized in the following tables:

Function value of $f(x) + g(x) = \|Ax - b\|$				
	5 iterations	50 iterations	100 iterations	200 iterations
ISTA	3.453e+05	2.662e+04	1.292e+04	5.535e+03
FISTA	2.033e+05	2.282e+03	4.887e+02	9.660e+01

Function value of $f(x) + g(x) = \|Ax - b\| + \|x\|_1$				
	5 iterations	50 iterations	100 iterations	200 iterations
ISTA	5.079e+06	4.766e+06	4.752e+06	4.745e+06
FISTA	4.940e+06	4.742e+06	4.740e+06	4.739e+06

It is clear that FISTA gives better results than ISTA, because the function values of FISTA are consistently lower than the function values of ISTA. Note that the function value of FISTA after only 50 iterations is better than the function value of ISTA after 200 iterations.

PSNR: peak signal to noise ratio for $f(x) + g(x) = \|Ax - b\|$				
	5 iterations	50 iterations	100 iterations	200 iterations
ISTA	2.172+01	2.367+01	2.446+01	2.533+01
FISTA	2.201+01	2.618+01	2.786+01	2.993+01

PSNR: peak signal to noise ratio for $f(x) + g(x) = \|Ax - b\| + \|x\|_1$				
	5 iterations	50 iterations	100 iterations	200 iterations
ISTA	2.171+01	2.368+01	2.448+01	2.535+01
FISTA	2.200+01	2.628+01	2.810+01	3.025+01

Here too, it is clear that FISTA gives better results than ISTA, because the PSNR of the FISTA method compared with the original image is consistently higher than the PSNR of the ISTA method compared with the original image.
Note that the PSNR of function $f(x)+g(x) = \|Ax-b\|+\|x\|_1$ is consistently higher than the PSNR of function $f(x) + g(x) = \|Ax - b\|$, so we can say that the l_1 regulator helps the iterations to converge faster.

5 Classification

In this section we discuss Stochastic Sub-Gradient Descent algorithms (SGD) for solving the optimization problem cast by Support Vector Machines.
SGD is a modification of the basic gradient descent algorithm, which allows us scaling these algorithms to much bigger training sets.
The problem with gradient descent is that if the number of the training examples is much bigger, computing the gradient could be very expensive or even impossible because each step requires storing all the data, and calculates the derivative. SGD doesn't need to look at all of the training set in every single iteration, but only at a single training example.
Support vector machine is a linear model for classification and regression problems that constructs a hyperplane that separates the data into classes. First, it randomly reorders the data examples, a pre-processing step which ensures that when scanning through the training set, the order of visiting the training examples would be in randomly sorted order, that speeds up the convergence.
Then it finds the points closest to the hyperplane from both the classes. These points are called support vectors. Then, the distance between the hyperplane and the support vectors is computed. This distance is called the margin. The goal is to maximize the margin, so the hyperplane for which the margin is maximized is the optimal hyperplane.
The main difference between gradient descent and SGC is that in SGC the algorithm modifies the parameters a little bit for every iteration on **one** training example to fit just the specific training example a little bit better.

Formally, given a training set $S = \{(\vec{x}_i, y_i)\}_{i=1}^{m}$ where $\vec{x}_i \in \mathbb{R}^n$ and $y_i \in \{+1, 1\}$, we would like to find the minimizer of the problem

$$\min_{\vec{w}} \frac{\lambda}{2}\|\vec{w}\|^2 + \frac{1}{m}\sum_{(\vec{x},y)\in S} l(\vec{w};(\vec{x},y)), \tag{8}$$

where

$$l(\vec{w};(\vec{x},y)) = max\{0, 1 - y\langle\vec{w},\vec{x}\rangle\}, \tag{9}$$

$\langle u, v\rangle$ denotes the standard inner product between the vectors u and v, λ is the regularization parameter of SVM, and l is called the Loss function.

We denote that

$$f(w) = \frac{\lambda}{2}\|\vec{w}\|^2 + \frac{1}{m}\sum_{(\vec{x},y)\in S} l(\vec{w};(\vec{x},y)),$$

hence, the gradient decent iterative algorithm in this case would be:

$$w_{t+1} = w_t - \eta_t \nabla f(w_t),$$

where $\eta_t > 0$ is called the learning rate. The gradient ∇f would be:

$$\nabla f(w) = \lambda\|w\| + \frac{\partial}{\partial w} max\{0, 1 - y\vec{w}\cdot\vec{x}\},$$

where the sub gradient is

$$\frac{\partial}{\partial w} = \begin{cases} -y_i x_i, & y_i\vec{w}\cdot\vec{x_i} < 1, \\ 0, & y_i\vec{w}\cdot\vec{x_i} = 1, \\ 0, & y_i\vec{w}\cdot\vec{x_i} > 1, \end{cases} \tag{10}$$

if $y_i\vec{w}\cdot\vec{x_i} < 1$ than

$$w_{t+1} = w_t - \eta_t(\lambda\|w_t\| - y_i x_i), \tag{11}$$

else $(y_i\vec{w}\cdot\vec{x_i} \geq 1)$ than

$$w_{t+1} = w_t - \eta_t\lambda\|w_t\|. \tag{12}$$

Any hyperplane can be written as the set of points $\vec{x}$ satisfying $\vec{w}\cdot\vec{x} - b = 0$ where $\vec{w}$ is the normal vector to the hyperplane. The following constraint were added to prevent data points from falling into the margin, for each i either:

$\vec{w}\cdot\vec{x}_i - b \geq 1$, if $y_i = 1$,

or

$\vec{w}\cdot\vec{x}_i - b \leq 1$, if $y_i = -1$.

These constraints state that each data point must lie on the correct side of the margin. It could also be written as:

$$y_i(\vec{w}\cdot\vec{x}_i - b) \geq 1, \quad for\ all\ \ 1 \leq i \leq n. \tag{13}$$

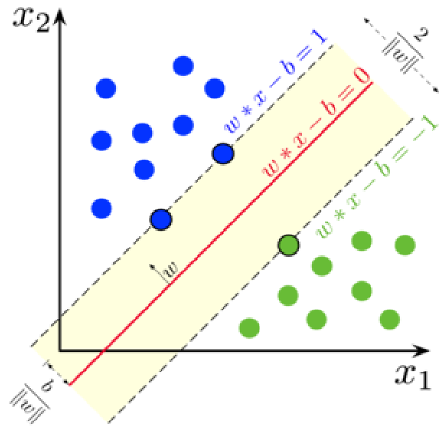


Figure 5: Maximum-margin hyperplane and margins for an SVM trained with samples from two classes

5.1 Pegasos

Pegasos performs stochastic gradient descent on the primal objective (8) with a carefully chosen stepsize. Pegasos gets as input S set of data examples, λ the regularization parameter of SVM and T the number of iterations.

```
**Input:** $S, \lambda, T$
**Initialize:** Set $w_1 = 0$
For $t = 1, 2, ..., T$
Choose $i_t \in \{1, ..., |S|\}$ uniformly at random.
Set $\eta_t = \frac{1}{\lambda t}$
If $y_{i_t}\langle w_t, x_{i_t}\rangle < 1$, then:
Set $w_{t+1} \leftarrow (1 - \eta_t\lambda)w_t + \eta_t y_{i_t} x_{i_t}$
Else (if $y_{i_t}\langle w_t, x_{i_t}\rangle \geq 1$):
Set $w_{t+1} \leftarrow (1 - \eta_t\lambda)w_t$
**Output:** $w_{T+1}$
```

On each iteration Pegasos operates in the following manner. Initially, it sets w_1 to the zero vector. On iteration t, it first chooses a random training example (x_{i_t}, y_{i_t}) by picking an index $i_t \in 1, ..., m$ uniformly at random. Then, it replaces the objective in (8) with an approximation based on the training example (x_{i_t}, y_{i_t}), yielding:

$$f(w; i_t) = \frac{\lambda}{2}\|w\|^2 + l(w; (x_{i_t}, y_{i_t})).$$

The sub gradient of the above approximate objective is given by:

$$\nabla_t = \lambda w_t - \mathbb{1}[y_{i_t}\langle w_t, x_{i_t}\rangle < 1] y_{i_t} x_{i_t},$$

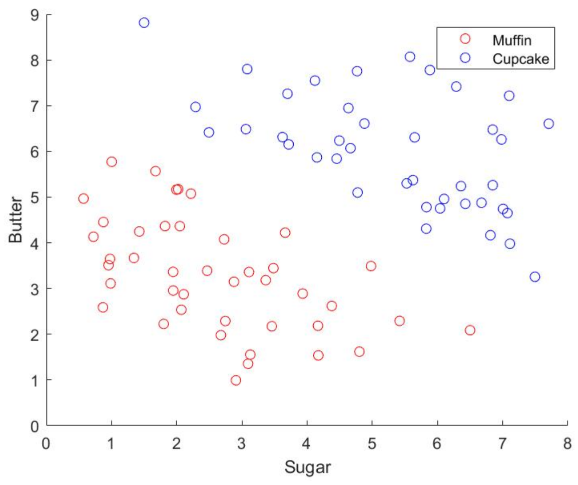
where $\mathbb{1}[y\langle w, x\rangle < 1]$ is the indicator function which takes a value of one if its argument is true (w yields non-zero loss on the example (x, y)), and zero otherwise. Then it updates $w_{t+1} \leftarrow w_t - \eta_t \nabla t$ using a step size of $\eta_t = \frac{1}{\lambda t}$. Note that this update can be written as:

$$w_{t+1} \leftarrow \left(1 - \frac{1}{t}\right) w_t + \eta_t \mathbb{1}\left[y_{i_t}\langle w_t, x_{i_t}\rangle < 1\right] y_{i_t} x_{i_t}\ .$$

After T iterations, the last iterate w_{T+1} is returned.

5.2 Numerical experiments

In this section we illustrate a data classification problem. We estimate a hyperplane $w^T x - b$ from a set of data and its labels $\{(\vec{x}_i, y_i)\}_{i=1}^m$ using the a built in SVM classifier in MATLAB and a stochastic gradient descent algorithm Pegasos. We compare those two algorithms and show the hyperplane each function finds.
In this example we would like to classify recipes as cupcakes or muffins. When given a new recipe, our model could determine if it's a cupcake or a muffin. We took 82 different recipes of cupcakes and muffins and normalized the data to percentage of the whole batter. Then, we chose 2 ingredients-butter and sugar that we've noticed that are different between those two types of recipes.



As you can see muffins have little sugar and little butter and cupcakes have a lot of sugar and a lot of butter, therefore it's a good example for SVM classify.
First, the data went through the SVM classifier (applied by the MATLAB function fitcsvm), the hyperplane and the support vectors solution for this problem are given in Figure (6).

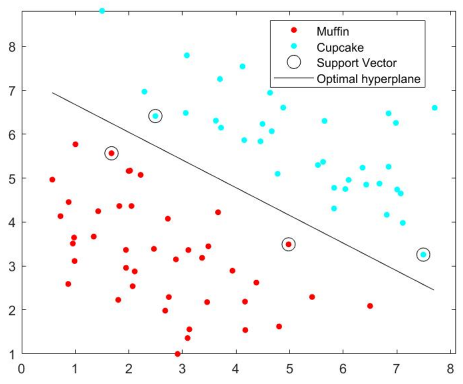


Figure 6: SVM hyperplane and support vector

The hyperplane defined by SVM is $(0.93, 1.47)x - 10.71$.
Then, we have implemented the Pegasos algorithm on MATLAB, the hyperplane and its margins are given in Figure (7)

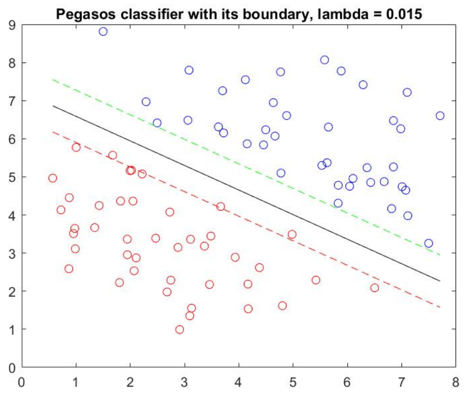
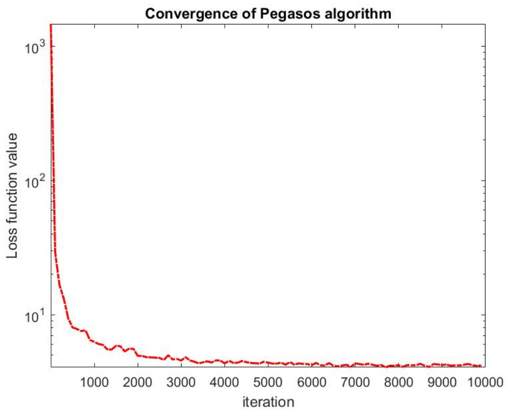


Figure 7: Pegasos hyperplane and its margins

The hyperplane defined by Pegasos is $(0.94, 1.46)x - 10.54$*.*
The results we got are approximately the same as the result of the built in SVM classifier function.
Another interesting information is the minimized function value, because we used a stochastic method we wont see a stable descent, but stochastic steps that their general target is the minimum.



6 Conclusion

In this project, we presented and described the gradient descent algorithm. We presented two extensions of the algorithm one at the field of data fitting and one on the field of classification.
We have shown that solving optimization problems is a common task in machine learning that arises at many different applications. For large-scale problems, those methods are often the only practical option, so extending those methods with better convergence rate could yield a tremendous influence at this field. We have also been able to analyze and implement the data fitting methods and classification method in an extremely simple and efficient way.

7 Acknowledgements

I would like to thank Assoc. Prof. Aviv Gibali for giving me the opportunity to work on this fascinating and instructive project. I would also like to thank him for the time and help he granted me to help me move forward with this project, while pushing me to be as independent as possible.

8 Matlab Codes

8.1 Data Fitting

FISTA Algorithm:

```
clear; close all; clc;

[A,b,m,n,Gray_Image]=Conv_Blur_Img('images.jpg');
[AT,ATA,Lipschitz,S_A] = Elements_Calc(A);
lambda = 1;
%Initial step
X0 = zeros(S_A(2),1); X02 = X0;
y0 = zeros(S_A(2),1); y02 = y0;
t0 = 1;

NumOfIter = 200;

for i = 1:NumOfIter
    %Iterative step
    Xk  = GradStep(Lipschitz,y0,AT,ATA,b);
    Xk2 = GradX1Step(Lipschitz,y02,AT,ATA,b,lambda);  %
        with norm1
    tk = 0.5*(1+sqrt(1+4*t0^2));
    yk=Xk+((t0-1)/tk)*(Xk-X0);
    yk2=Xk2+((t0-1)/tk)*(Xk2-X02);

    y0=yk;     y02=yk2;     t0=tk;     X0=Xk;     X02=Xk2;

    if i==5
        Img_Per_Iter(Xk,Xk2,m,n,Gray_Image,i)
        Fi(A,Xk,Xk2,b,i)
    elseif i==50
        Img_Per_Iter(Xk,Xk2,m,n,Gray_Image,i)
        Fi(A,Xk,Xk2,b,i)
    elseif i==100
        Img_Per_Iter(Xk,Xk2,m,n,Gray_Image,i)
        Fi(A,Xk,Xk2,b,i)
    end
```

```
end

Img_Per_Iter(Xk,Xk2,m,n,Gray_Image,i)
Fi(A,Xk,Xk2,b,i)
```

ISTA Algorithm:

```
clear; close all; clc;

[A,b,m,n,Gray_Image]=Conv_Blur_Img('images.jpg');
[AT,ATA,Lipschitz,S_A] = Elements_Calc(A);
lambda = 1;
%Initial step
X0 = zeros(S_A(2),1);   X02 = X0;
NumOfIter = 200;

for i = 1:NumOfIter
    %Iterative step
    Xk  = GradStep(Lipschitz,X0,AT,ATA,b);
    Xk2 = GradX1Step(Lipschitz,X02,AT,ATA,b,lambda);   %
       with norm1
    X0=Xk;      X02=Xk2;
    if i==5
        Img_Per_Iter(Xk,Xk2,m,n,Gray_Image,i)
        Fi(A,Xk,Xk2,b,i)
    elseif i==50
        Img_Per_Iter(Xk,Xk2,m,n,Gray_Image,i)
        Fi(A,Xk,Xk2,b,i)
    elseif i==100
        Img_Per_Iter(Xk,Xk2,m,n,Gray_Image,i)
        Fi(A,Xk,Xk2,b,i)
    end
end

Img_Per_Iter(Xk,Xk2,m,n,Gray_Image,i)
Fi(A,Xk,Xk2,b,i)
```

Image Blur:

```
%This function reads an image, blur it via convolution
```

```
%and return this action as a system of linear equations
    .
function [A,b,m,n,Image]=Conv_Blur_Img(image_name)
%Convert to gray image
Image=double(rgb2gray(imread(image_name)));
%Generate a Gaussian kernel of size 9*9 std=4, and
    convulate it with the image
h = fspecial('gaussian',9,4);
Blur_Img=conv2(h,Image);
%reshape a matrix as a vector column
m=size(Image,1);
n=size(Image,2);
p=size(Blur_Img,1);
q=size(Blur_Img,2);
b=double(reshape(Blur_Img,p*q,1));
%Generate a matrix operator of Blurr
A=convmtx2(h,m,n);

subplot(1,2,1)
imshow(uint8(Image))
title('Original Image');
subplot(1,2,2)
imshow(uint8(Blur_Img))
title('Blurred Image')
```

Elements Calculator:

```
function [AT,ATA,Lipschitz,S_A] = Elements_Calc(A)
AT = A.';
ATA = AT*A;
S_A=size(A);
Lipschitz = 2*eigs(ATA,1);    %%Lip = 2*max_eighenvalue
end
```

Gradient step:

```
function Xk=GradStep(Lipschits,X0,AT,ATA,b)
        Xk= X0-(1/Lipschits)*(2*ATA*X0-2*AT*b);
end
```

Gradient step with norm1:

```
function Xk=GradX1Step(Lipschits,X0,AT,ATA,b,lambda)
        Xk = X0-(1/Lipschits)*(2*ATA*X0-2*AT*b+lambda*
           sign(X0));
end
```

Show image in current iteration:

```
function Img_Per_Iter(Xk,Xk2,m,n,Image,i)
    figure
    subplot(1,2,1);
    imshow(uint8(reshape(Xk,m,n)));
    t =  sprintf('PSNR after %d ietrations: ',i);
    str1 = sprintf('|Ax-b|= %d',psnr(uint8(reshape(Xk,m
       ,n)),uint8(Image)));
    title({t;str1})

    subplot(1,2,2);
    imshow(uint8(reshape(Xk2,m,n)));
    str2 = sprintf('|Ax-b|+|x|_1= %d',psnr(uint8(
       reshape(Xk2,m,n)),uint8(Image)));
    title(str2)

end
```

Value of function in current iteration:

```
function Fi(A,Xk,Xk2,b,i)
fprintf('Iteration: %d, |A*X-b|= %d',i,norm(A*Xk-b)^2);
fprintf(', |A*X-b|+|x|_1= %d\n',norm(A*Xk2-b)^2 + norm(
   Xk2,1));
end
```

8.2 Classification

main function SVM VS Pegasos:

```
clear; close all; clc;

data = xlsread('data');
% define variables
```

```
X = data(:,4:5);
y = data(:,6);

T=10000;
lambda = 0.015;

mdl = SVM_solver(X,y);
[wT,loss,b]=Pegasos_solver(X,y,lambda,T);
```

SVM solver:

```
function [mdl]=SVM_solver(X,y)
%%%%%%fit SVM model%%%%%%%%%%%%%%%%%%%%%%%%
mdl = fitcsvm(X,y);
sv = mdl.SupportVectors;
%%%%%%presenting results of svm model%%%%%%
figure
gscatter(X(:,1),X(:,2),y)
hold on
plot(sv(:,1),sv(:,2),'ko','MarkerSize',10)
d = 0.02;
[X1Grid,X2Grid] = meshgrid(min(X(:,1)):d:max(X(:,1)),
    min(X(:,2)):d:max(X(:,2)));
XGrid = [X1Grid(:),X2Grid(:)];
[~,scores] = predict(mdl,XGrid);
contour(X1Grid,X2Grid,reshape(scores(:,2),size(X1Grid))
    ,[0 0],'k');
legend('Cupcake','Muffin','Support Vector','Optimal
    hyperplane')
fprintf('The hyperplane defined by SVM is [(%.2f,%.2f)X
     %.2f\n',mdl.Beta,mdl.Bias);
end
```

Pegasos solver:

```
function [w,lossval,b]=Pegasos_solver(X,y,lambda,T)
[n,~] = size(X);
%%%initial values%%%
w = zeros(2,1);
lossval=zeros(T,1);
```

```
b=sum(y-X*w)/n;
for t= 1:T

    eta = 1/(lambda*t);
    permi = randperm(n);
    for i = 1:n
        j=permi(i);
        if y(j)*(dot(w,X(j,:)')+b) < 1
            w = (1-eta*lambda)*w + eta*y(j)*X(j,:)';
            b = b + eta*y(j);
        else
            w = (1-eta*lambda)*w;
        end
    end
    l=max(0,1 - (y.*(X*w)));
    lossval(t) = 0.5*lambda*w'*w + (1/n)*sum(l);
end

figure
step = 100;
semilogy( 1:step:T, lossval(1:step:end), 'r-.','
   LineWidth',1.5 );
xlabel('iteration');
ylabel('Loss function value');
axis([1 T 0 max(lossval)]);
title('Convergence of Pegasos algorithm');

% visualize
figure
xp  = linspace(min(X(:,1)), max(X(:,1)));
yp  =- (w(1)*xp +b     )/ w(2);
yp1 =- (w(1)*xp +b - 1)/ w(2); % margin boundary for
   support vectors for y=1 (muffin)
yp0 =- (w(1)*xp +b + 1)/ w(2); % margin boundary for
   support vectors for y=-1 (cupcake)

i_cupcake = find(y==-1);% index of cupcake samples
i_muffin = find(y==1);% index of muffin samples
```

```
plot(X(i_cupcake,1), X(i_cupcake,2), 'ro');
hold on
plot(X(i_muffin,1), X(i_muffin,2), 'bo');
plot(xp, yp, '-k' , xp, yp1, '--g', xp, yp0, '--r');
title(sprintf('Pegasos classifier with its boundary,
   lambda = %g', lambda));

fprintf('The hyperplane defined by Pegasos is (%.2f,%.2
   f)X %.2f\n',w,b);
end
```

References

[1] Shalev-Shwartz, Shai, et al. ”Pegasos: Primal estimated sub-gradient solver for svm.” Mathematical programming 127.1 (2011): 3-30.

[2] Beck, Amir, and Marc Teboulle. ”A fast iterative shrinkage-thresholding algorithm for linear inverse problems.” SIAM journal on imaging sciences 2.1 (2009): 183-202.

[3] Beck, Amir. Introduction to nonlinear optimization: Theory, algorithms, and applications with MATLAB. Vol. 19. Siam, 2014.

[4] https://see.stanford.edu/materials/lsoeldsee263/05-ls.pdf

[5] https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm