Coordinate Descent vs. (Accelerated) Proximal Gradient for Lasso

Marcus Artner, Hannes Janker & Lukas Till Schawerda
Optimization for Data Science, University of Vienna

***Abstract*—First order methods like gradient descent are a crucial part for solving optimization problems. These methods, using the gradient of an objective function, play an essential role in a large number of machine learning and data science tasks. The basic framework of these algorithms is to approximate the problem by an easier model, solve the model and iteratively continuing this procedure. Applying these methods is often times much computationally cheaper than computing closed form solutions for the problems. In the following, two gradient methods are described and compared for the Lasso objective function.**

I. Introduction

The aim of this project is to introduce and compare the two gradient methods coordinate descent and proximal gradient descent. These algorithms will be applied to the Lasso objective function. Firstly, both of the methods will be briefly explained. Moreover, a short introduction about the Lasso problem will be given. In the main part, the performance of the algorithms will be compared with implementations in Python. Besides that, it will be tried to achieve acceleration for both of the algorithms. Furthermore, the learning rate decreases with the epoch by a specified factor. In the final part, these convergence rates will be plotted and the performances will be compared.

II. Theory about the Algorithms and Lasso

Coordinate descent is an optimization algorithm that minimizes along the coordinate direction to minimize a function. Via a selection rule, the algorithm chooses one coordinate in each iteration and minimizes over the coordinate while all the other coordinates remain fixed. In other words, only one coordinate is modified per iteration. The algorithm basically contains two steps per iteration: select $i_k \in \{1, ..., d\}$ and calculate

$$x_{k+1} = x_k + \gamma * e_{i_k},$$

where ei_k is the i-th unit basis vector. There are two main variants. The first one chooses gradient-based step sizes:

$$x_{k+1} = x_k - \frac{1}{L}\nabla_{i_k} f(x_k) * e_{i_k}$$

The second one computes the exact coordinate minimization by solving the scalar problem:

$$argmin_{\gamma \in R} f(x_k + \gamma * e_{i_k})$$

While the first one needs line search for the step sizes the second one is hyperparameter free. There are two technical assumption that we make. We assume strong convexity and we assume coordinate-wise L-smoothness, i.e.

$$f(x + \gamma * e_i) \leq f(x) + \gamma \nabla_i f(x) + \frac{L}{2}\gamma^2$$

$\forall x \in R^d, \forall \gamma \in R, \forall i \in [d]$

There are several ways for selecting the coordinates. The most naive approach would be to simply cycle through all coordinates. A common practice is to select the coordinates uniformly at random ($i_k \in \{1, ..., d\}$). Other ways of choosing the coordinates would be importance sampling and steepest coordinate descent.
Proximal gradient methods can be seen as a generalized form of projection which is used to solve non-differentiable convex optimization problems. Consider an objective function that can be composed as

$$f(x) = g(x) + h(x)$$

where g is a convex and differentiable function and h is a simple function which is convex but not necessarily differentiable. For the proximal gradient algorithm the proximal mapping for a function h and parameter α needs to be defined.

$$prox_{\alpha h}(x) = argmin_{y \in R}\{h(y) + \frac{1}{2\alpha}||y - x||^2\}$$

An iteration of the algorithm looks as following:

$$x_{k+1} = prox_{\alpha h}(x_k - \alpha \nabla g(x_k))$$

The proximal mapping $prox_\alpha(\cdot)$ can be computed analytically for many important functions h and it depends only on h, not on g. This means even if g is a complicated function, only its gradient needs to be computed. The computational cost of $prox_\alpha(\cdot)$ depends of course on h. An important application is the Lasso objective which includes a non-smooth regularizer term.

The Lasso regression is a type of linear regression that uses shrinkage which means that the coefficients of the estimator are shrunk towards a central point. Lasso encourages simple, sparse models and is well-suited for models with high multicollinearity or for automatic model selection parts. The objective function which should be minimized is a quadratic and looks the following:

$$F(\beta) = \frac{1}{2}||Y - X\beta||_2^2 + \lambda||\beta||_1$$

$\lambda||\beta||_1$ is called regularizer term. This kind of regularization can result in sparse models with few parameters and some coefficients can even become zero and will be eliminated from the model. How strong the so called penalty is, depends on the tuning parameter λ. The gradient of the objective function is given by

$$\frac{d}{d\beta}F(\beta) = X^T(Y - X\beta) + \lambda\,\text{sign}(\beta)$$

with the sign$(\cdot)$-function being applied componentwise. Note that the gradient of the $||\cdot||_1$-norm at the point 0 is thus set to 0.

In this setting, coordinate descent yields the updating scheme

$$\beta^j_{k+1} = \beta^j_k + \alpha X^T_{\cdot j}(Y - X\beta) + \lambda\,\text{sign}(\beta^j_k)$$

where β^j_k is the j-th component of β_k and $X_{\cdot j}$ is the j-th column of X. During each epoch, the order in which the components are updated is determined at random. If an acceleration parameter is specified, then Nesterov momentum is applied after each epoch.

Proximal gradient descent for the Lasso problem follows the updating scheme

$$\beta_{k+1} = prox_\alpha(\beta_k + \alpha X^T(Y - X\beta))$$

with

$$[prox_\alpha(\beta)]_j = \begin{cases} \beta^j + \lambda\alpha & \text{if } \beta^j < -\lambda\alpha \\ 0 & \text{if } -\lambda\alpha \leq \beta^j \leq \lambda\alpha \\ \beta^j - \lambda\alpha & \text{if } \lambda\alpha < \beta^j \end{cases}$$

III. Implementation

We implemented the updating schemes explained above in Python using the $numpy$ library. All plots were creating using $matplotlib.pyplot$. The functions $coordinate\_descent()$ and $proximal\_gradient\_descent()$ take the following inputs:

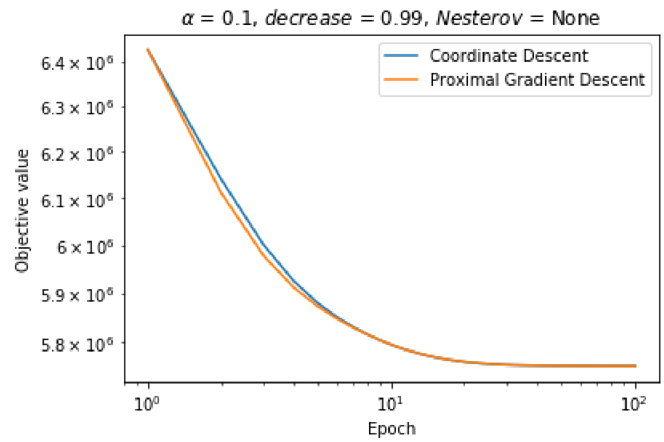
PARAMETER	INPUT
X	Data
y	Targets
beta_0	Starting point
alpha	Learning rate
num_epochs	Number of epochs
decr	Factor for decreasing learning rate
accelerate	Nesterov acceleration parameter

They output both the objective value over the course of learning and the final estimate for β. Auxiliary functions for computing intermediate results were also implemented, but are not explained in detail here. For further information one can consider the code.

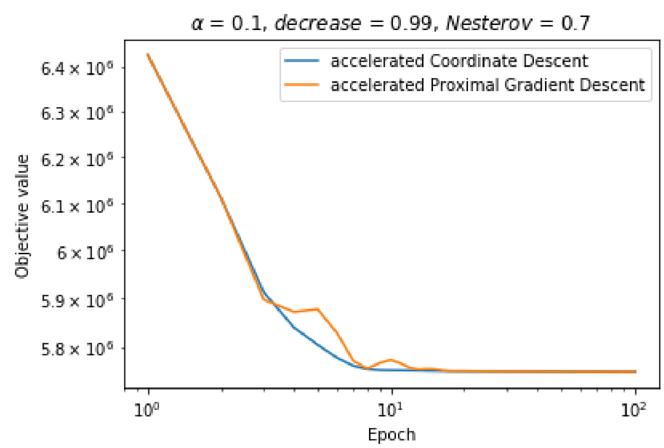
The $california\_housing$-dataset from sklearn was used for evaluation. It has 20640 observations and 8 parameters.

IV. Results

The initial weight vector β_0 was set to all zeros. First we compare coordinate descent and proximal gradient descent without acceleration.



We observe that Coordinate Descent, very similar to Proximal Gradient Descent converges rather fast after about 20 epochs. Neither algorithm performs significantly better than the other.



If we accelerate both algorithms with a Nesterov momentum parameter of 0.7, we observe slightly faster convergence over the not accelerated algorithms. Accelerated proximal gradient descent shows signs of overshooting in the first steps due to momentum.

Overall, both methods display excellent performance. This holds true even for various other data sets that we tried. Both methods are thus appropriate for solving the Lasso problem. We cannot conclude that one method is better than the other.

V. Summary

To sum up, we implemented both, coordinate descent and proximal gradient descent for solving optimization problems. In particular, we applied the algorithms to the Lasso objective function using open source data sets like the $california\_housing$-dataset. We were able to achieve a small acceleration by implementing Nesterov's momentum. Our final conclusion is that none of the both algorithms can be considered better based on our findings.

VI. Appendix: Important parts of the code

```
# Define Objective function

def objective(X,y,beta,lam):
    obj = (1/2)*np.linalg.norm(y - X@beta, 2)**2 +
     lam*np.linalg.norm(beta, 1)
    return obj


###### Coordinate Descent

def compute_coordinate_grad(X,y,beta,coordinate,lam)
    :
    """
    INPUT
    X...Data Matrix
    Y...Targets
    coordinate...direction which will be optimized
    lam...lambda
    """

    # Dimensions
    n,p = X.shape

    # j-th column of X
    x_j = X[:,coordinate]

    # j-th element of beta
    beta_j = beta[coordinate]

    # Regularizer
    reg = lam*np.sign(beta_j)

    # Compute gradient
    grad = -x_j@(y-X@beta) + reg

    # Return results
    return grad

def coordinate_descent(X, y, beta_0, lam, alpha,
     num_epochs, decr = None, accelerate = None):
    """
    INPUT
    X...Data Matrix
    Y...Targets
    beta_0...initial beta vector
    alpha...learning rate
    num_epochs...number of epochs
    decr...Parameter to decrease learning rate in
     each epoch
    accelerate...Nesterov acceleration parameter
    """
    # Get Dimensions of data and prepare beta values
      for acceleration

    n,p = X.shape
    beta = beta_0.copy()
    beta_old = beta_0.copy()

    # Initialize empty objective list

    obj = []
    obj.append(objective(X, y, beta_0, lam))

    # Iterate over epochs

    for epoch in range(num_epochs):

        # Sample coordinates for current epoch

        ind = np.random.permutation(p)

        if accelerate is not None:
```

```
            # Nesterov if accelerate is True

            beta_new = beta.copy()
            beta_temp = beta + accelerate * (beta - beta_old)

            for j in ind:

                # Compute gradient at accelerated point

                grad = compute_coordinate_grad(X, y, beta_temp, coordinate = j, lam = lam)

                # update j-th component of beta

                beta[j] = beta_temp[j] - alpha*grad
            beta_old = beta_new.copy()

        else:

            #regular Coordinate descent

            for j in ind:

                grad = compute_coordinate_grad(X, y, beta, coordinate = j, lam = lam)
                beta[j] -= alpha*grad

        # append current objective

        obj.append(objective(X,y,beta,lam))

        # Adapt learning rate if decr = True

        if decr is not None:

            alpha *= decr

    return obj, beta


##### Proximal Gradient Descent


# Soft Threshold Operator

def soft_thresh(beta, lam):
    """
    INPUT
    beta...Vector beta
    lam...lambda
    """

    res = np.zeros_like(beta)
    p = len(beta)

    # adjust beta according to formula

    for i in range(p):
        if beta[i] < -lam:
            res[i] = beta[i] + lam
        elif np.abs(beta[i]) < lam:
            res[i] = 0
        elif beta[i] > lam:
            res[i] = beta[i] - lam
    return res


def proximal_gradient_descent(X, y, beta_0, lam, alpha, num_epochs, decr = None, accelerate = None):
    """
    INPUT
    X...Data Matrix
    Y...Targets
    beta_0...initial beta vector
    alpha...learning rate
    num_epochs...number of epochs
    decr...Parameter to decrease stepsize in each epoch
    accelerate...Nesterov acceleration parameter
    """

    n,p = X.shape
    beta = beta_0.copy()
    beta_old = beta_0.copy()

    # initialize objective list

    obj = []
    obj.append(objective(X, y, beta_0, lam))

    # Iterate over epochs

    for epoch in range(num_epochs):

        # Nesterov if accelerate = True

        if accelerate is not None:

            beta_new = beta.copy()
            beta_temp = beta + accelerate * (beta - beta_old)
            content = beta_temp + alpha*X.T@(y - X@beta)
            beta = soft_thresh(content, lam*alpha)
            beta_old = beta_new.copy()

        # updating beta with soft threshold operator according to formula

        else:
            content = beta + alpha*X.T@(y - X@beta)
            beta = soft_thresh(content, lam*alpha)

        # append cunrrent objective

        obj.append(objective(X, y, beta, lam))

        # adapt learnign rate if decr = True

        if decr is not None:
            alpha *= decr

    return obj, beta
```