Homework 4

Due April 30th 2020 by 11:59pm

Instructions: This homework consists of a reading assignment and one coding exercises. Please submit your solutions via Gradescope. Solutions should consist of three files: a PDF containing your solutions to the *non-coding questions*; a Jupyter notebook (`.ipynb` file) and an html print-out (`.html` file) with your solution to the *coding exercise.* **All coding exercises must be completed in Python.** Please be sure to comment the code appropriately. Students are encouraged to discuss homework problems, particularly on Canvas and in the TA hours, but must submit their own solutions.

Reading Assignment

- Review Lecture 4.
- Review Lab 4 (available from the Course Materials page on Canvas).
- Read Section 5.6 in *Mathematics of Machine Learning.*

Coding

Please submit your solutions the coding exercise below as a Jupyter notebook (`.ipynb` file) and an html print-out (`.html` file) under the `Homework 4 - Coding` assignment in Gradescope. **Please run all cells in your notebook prior to submission, so we can view their output.**

1 Exercise 1

In this exercise, you will implement in **Python** a first version of *your own coordinate descent algorithm* to solve the LASSO problem, that is the ℓ_1-regularized least-squares regression problem.

Recall from the lectures that the LASSO problem writes as

$$\min_{\beta\in\mathbb{R}^d} F(\beta) := \frac{1}{2n}\sum_{i=1}^{n}(y_i - x_i^T\beta)^2 \ + \lambda\|\beta\|_1 \ . \tag{1}$$

Algorithm 1 Coordinate Descent Algorithm, general form

```
initialization   $\beta = 0$.
repeat for $t = 0, 1, 2, \dots$
    • Pick a coordinate index $j$ in $\{1, \dots, d\}$
    • Find $\beta_j^{\text{new}}$ by minimizing $F(\beta)$ with respect to $\beta_j$ only.
    • Perform update $\beta_j^{(t+1)} = \beta_j^{\text{new}}$.
    • Perform update $\beta_\ell^{(t+1)} = \beta_\ell^{(t)}$ for all $\ell \neq j$.
until the stopping criterion is satisfied.
```

Coordinate Descent

The coordinate descent algorithm is outlined in Algorithm 1.

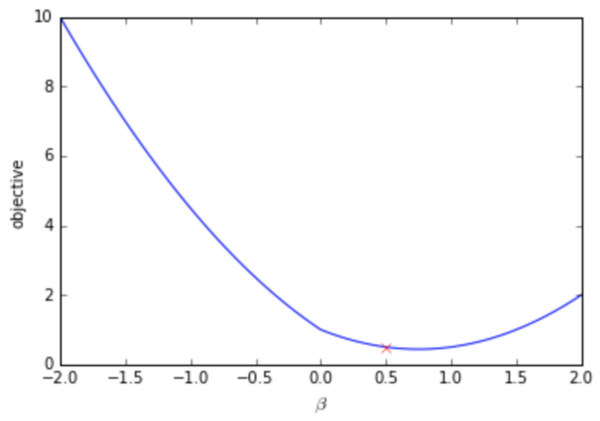
(a) Assume that $d = 1$ and $n = 1$. The sample is then of size 1 and boils down to just (x, y); the learning parameter β is then a scalar. The function F writes simply as

$$F(\beta) = \frac{1}{2}(y - x\,\beta)^2 \; + \lambda|\beta| \; . \tag{2}$$

The solution to the minimization problem $\min_\beta \; F(\beta)$ is given by soft thresholding:

$$\beta = \begin{cases} \frac{xy-\lambda}{x^2} & xy > \lambda \\ 0 & |xy| \leq \lambda \\ \frac{xy+\lambda}{x^2} & xy < -\lambda \end{cases} .$$

Implement the function $F(\beta)$ in the $n = 1$, $d = 1$ case. Implement a soft thresholding function that returns β given x, y, λ. Plot $F(\beta)$, setting $x = 1, y = 1, \lambda = 0.5$, and show visually that the minimum is obtained by your soft thresholding function. The result should look like this:



(b) Assume now that $d > 1$ and $n > 1$. The full minimization problem now writes as

$$F(\beta) := \frac{1}{2n} \sum_{i=1}^{n} (y_i - x_i^T \beta)^2 \ + \lambda \|\beta\|_1 \, . \tag{3}$$

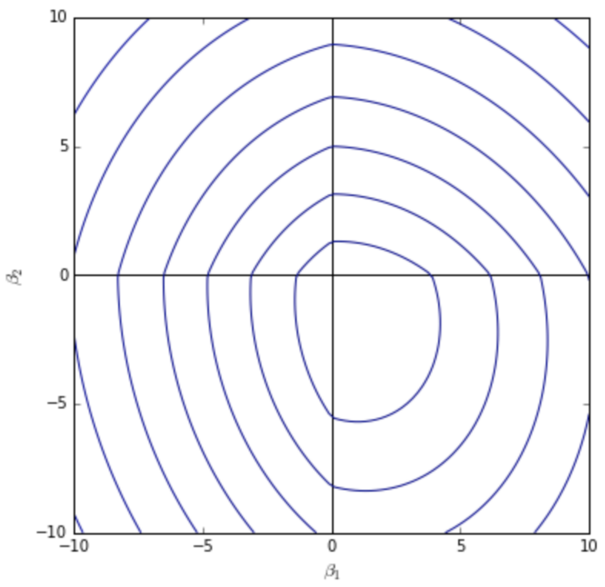
Write a function *computeobj* that computes and returns $F(\beta)$ for any β, given X, y, and λ. Plot this function by first generating some data:

```
np.random.seed(123)

X = np.random.normal(size=(10, 2))
beta = np.array([2.0, -5.0])
y = np.dot(X, beta) + np.random.normal(10)
lam = 3.0
```

and then passing your implementation of *computeobj* as an argument to the plotting function provided below:

```
def plot_objective(X, y, computeobj, lambda_, min_beta=None, iterates=None):
    nb = 100
    brange = np.linspace(-10, 10, nb)
    b1, b2 = np.meshgrid(brange, brange)

    z = np.array([computeobj(beta, X, y, lambda_) for
                  beta in zip(b1.ravel(), b2.ravel())])

    levels=np.logspace(-5,4,100)

    plt.figure(figsize=(6,6))

    plt.hlines(y=0, xmin=-10, xmax=10, color='k')
    plt.vlines(x=0, ymin=-10, ymax=10, color='k')

    if min_beta is not None:
        plt.scatter(min_beta[0], min_beta[1], marker="x", s=100, color='k')

    f iterates is not None:

        # Arrows.
        for j in range(1, len(iterates)):
            plt.annotate(
                "",
                xy=iterates[j],
                xytext=iterates[j - 1],
                arrowprops={"arrowstyle": "->", "color": color, "lw": 1},
                va="center",
                ha="center",
            )

    plt.contour(brange, brange, z.reshape((nb, nb)), levels=levels);
    plt.show()
```

The result should look like this:



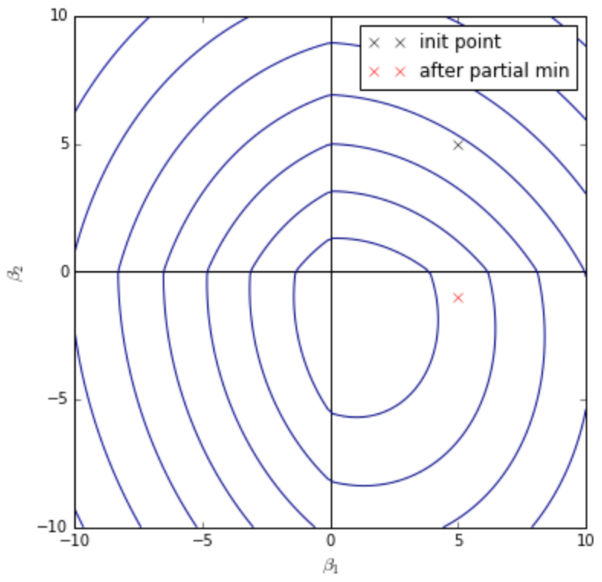
(c) Coordinate descent proceeds by sequential partial minimization with respect to each coordinate β_j, that by solving partial minimization problems of form

$$\min_{\beta_j} \quad \frac{1}{2n}\sum_{i=1}^{n}\{y_i - (\beta_1 x_{i,1} + \cdots + \beta_j x_{i,j} + \cdots + \beta_d x_{i,d})\}^2 \\ + \lambda\{|\beta_1| + \cdots + |\beta_j| + \cdots + |\beta_d|\} \ .$$

Write a function *partial_min* that takes X, y, β, λ and an index j, and returns the vector $\tilde{\beta}$ that results from minimizing $F(\beta)$ along coordinate j, starting at the point β. Using the same X, y, λ as above, evaluate this function from the starting point `beta_init` and on coordinate $j = 1$:

```
beta_init = np.array([5, 5])
j = 1
```

Generate the same contour plot as above, this time adding the initial point and the point that results from your partial minimization. The result should look like this:



(d) Download the file `HW4_data.csv` from the course website. The data is a subset of the `superconductivity` dataset from the UCI Machine Learning Repository. It consists of $n = 100$ observations of $d = 20$ features and one response. The goal is to predict the response, `critical_temp`, from the features. Load the data as a Pandas data frame, separating it into features X and response y. Standardize X and center y.

(e) Use the sklearn function `LassoCV` to find a suitable choice for the regularization parameter λ, with

```
alphas = np.logspace(-2,2,31)
```

and using the keyword parameter `cv=5` to specify 5-fold cross-validation. Report the best-performing choice of λ. Repeat the question for n-fold cross-validation that is *leave-one-out* cross validation.

(f) We will implement two versions of coordinate descent, corresponding to two methods for selecting the index j. First, write a function *cycliccoorddescent* that implements the *cyclic coordinate descent* algorithm. The cyclic version of this algorithm selects j by starting at $j = 0$ and incrementing it sequentially (modulo d) in each iteration. Thus we first minimize with respect to the coordinate β_0, then β_1, and so on. Set a maximum iteration number as the stopping criterion.

(g) Next write a function (or modify the previous function such that it can perform either selection method) such that the index j is chosen uniformly at random. You may wish to use your solution to part (c).

(h) Run both implementations on the data X, y using the value of λ selected previously. Set the maximum number of iterations to 20000. For both cyclic and random coordinate descent, plot the Lasso objective as a function of the iteration number.

(i) Run both implementations, *cycliccoorddescent* and *randcoorddescent*, on the data X, y using the value of λ selected previously either by 5-fold or leave-one-out cross-validation. Set the maximum number of iterations to 20000. For both variants, cyclic and randomized, of coordinate descent, plot the Lasso objective as a function of the iteration number.

(j) Compare the final iterate of each method to the result of sklearn's `Lasso`.

(k) Comment on the sparsity of the result. Which features are selected by your coordinate descent method?

(l) Plot the regularization paths for the randomized variant of coordinate descent. Comment on the regularization paths.