Machine Learning homework 5 solution
Optimization

Wiktor Jurasz - M.Nr. 03709419

November 22, 2018

1 Problem 1

1.1

We know that sum of convex functions is also convex, so in this case it's enough to prove that each component is convex.

- x^2 - Convex because second derivative (2) is always non negative
- $2y$ - Convex because linear functions are convex.
- $cos(sin(\sqrt{\pi}))$ - Convex because constant are convex (it's also just linear function).
- $-\min\{-x^2, log(y)\} = \max(x^2, -log(y))$ - Convex because max of convex functions is convex (proven later). $-log(y)$ is convex because second derivative $\frac{1}{x^2}$ is always non negative (and logarithm is defined on $(1, 50)$.

$f(x, y)$ **is convex**

1.2

$$f'(x) = x^{-1} - 3x^2 \tag{1}$$
$$f''(x) = -x^{-2} - 6x \tag{2}$$

$$\begin{gathered} f''(x) \geq 0 \Rightarrow \\ -x^{-2} - 6x \geq 0 \Rightarrow \\ -1 - 6x^3 \geq 0 \Rightarrow \\ -1 \geq 6x^3 \Rightarrow \\ x \leq \sqrt[3]{\frac{-1}{6}} \Rightarrow \\ x \leq\sim -0.55 \end{gathered} \tag{3}$$

$(-\infty, -0.55) \cap (1, \infty) = \emptyset \Rightarrow f(x)$ **is not convex**

1.3

$$f(x) = -\min\{log(3x+1), -x^4 - 3x^2 + 8x - 42\} = \max\{-log(3x+1), x^4 + 3x^2 - 8x + 42\} \tag{4}$$

The first part of the max is convex because $-log$ is convex and non-decreasing and $3x + 1$ is convex and composition of convex non-decreasing function with convex function is convex.
The second part of max is convex because all elements of the polynomial are convex.
Since both functions are convex and max of convex functions is convex $f(x)$ **is convex**

1.4

$$\begin{aligned}
\frac{\partial f}{\partial x} &= 3yx^2 - 2yx \\
\frac{\partial f}{\partial y} &= x^3 - x^2 + 2y + 1 \\
\frac{\partial f}{\partial x^2} &= 6yx - 2y \\
\frac{\partial f}{\partial x \partial y} &= 3x^2 - 2x \\
\frac{\partial f}{\partial y \partial x} &= 3x^2 - 2x \\
\frac{\partial f}{\partial y^2} &= 2
\end{aligned} \tag{5}$$

$$M = \begin{bmatrix} 6yx - 2y & 3x^2 - 2x \\ 3x^2 - 2x & 2 \end{bmatrix}$$

$$\begin{bmatrix} a & b \end{bmatrix} M \begin{bmatrix} a \\ b \end{bmatrix} = a^2(6yx - 2y) + 2ab(3x^2 - 2x) + 2b^2 \tag{6}$$

For function to be convex above scalar has to be non-negative for any natural a and b.
However, following values:

$$\begin{aligned}
a &= 1 \\
b &= 0 \\
x &= 0 \\
y &= 1
\end{aligned} \tag{7}$$

gives us $1 * -2 * 1 = -1 < 0 \Rightarrow f(x, y)$ **is not convex**

2 Problem 2

Let's take:

$$\begin{aligned}
x, y &\in \mathbb{R}^d \\
\lambda &\in (0, 1)
\end{aligned}$$

Then:

$$\lambda x + (1 - \lambda)y$$

Also belongs to convex set of f_1, f_2.

$$\begin{aligned}
h(\lambda x + (1-\lambda)y) &= \max\{f_1(\lambda x + (1-\lambda)y), f_2(\lambda x + (1-\lambda)y\} \\
&\leq \max\{\lambda f_1(x) + (1-\lambda)f_1(y), \lambda f_2(x) + (1-\lambda)f_2(y)\} \\
&\leq \lambda \max\{f_1(x), f_2(x)\} + (1-\lambda)\max\{f_1(y), f_2(y)\} \\
&= \lambda h(x) + (1-\lambda)h(y)
\end{aligned} \tag{8}$$

Q.E.D

3 Problem 3

$$g'(x) = f_1'(f_2(x))f_2'(x) \tag{9}$$

$$g''(x) = f_1''(f_2(x))(f_2'(x))^2 + f_1'(f_2(x))f_2''(x) \tag{10}$$

From the fact that f_1, f_2 are convex and square property we can derive following:

$$\begin{aligned} f_1'' &\geq 0 \\ (f_2'(x))^2 &\geq 0 \\ f_2''(x) &\geq 0 \end{aligned} \tag{11}$$

For g to be convex $g''(x)$ has to be non-negative. However, $f_1'(f_2(x))$ can be both negative and positive. Now if we assume additional properties for f_1

- f_1 is decreasing on $\mathbb{R}$
- $f_1''(x) = 0$

(e.g $f_1(x) = -x$)
Then $g''(x) < 0 \Rightarrow f_1(f_2(x))$ **is not (always) convex**

4 Problem 4

Assumptions

- f is convex
- $x, y \in \mathbb{R}^n$
- $\nabla f(x) = 0, \nabla f(y) = 0$ are local minima
- $f(x) \geq f(y)$
- $\lambda \in (0, 1)$

From assumptions and convexity definition:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \leq \lambda f(x) + (1-\lambda)f(x) = f(x) \tag{12}$$

To write it more clearly:

$$f(\lambda x + (1-\lambda)y) \leq f(x) \tag{13}$$

Because of our assumptions regarding λ we know that $\lambda x + (1-\lambda)y$ lies somewhere "between" x and y. But we also said that $f(x)$ is a local minimum, which means it has to be smaller then all the values in it's neighbourhood. So the only way to satisfy equation 13 is to set $x = y$ which means that there is only one (global) minimum.

5 Programming assignment 5

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        import math
        %matplotlib inline

        from sklearn.datasets import load_breast_cancer
        from sklearn.model_selection import train_test_split
        from sklearn.metrics import accuracy_score, f1_score
```

5.1 Your task

In this notebook code skeleton for performing logistic regression with gradient descent is given. Your task is to complete the functions where required. You are only allowed to use built-in Python functions, as well as any `numpy` functions. No other libraries / imports are allowed.

For numerical reasons, we actually minimize the following loss function

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} NLL(\mathbf{w}) + \frac{1}{2}\lambda ||\mathbf{w}||_2^2$$

where $NLL(\mathbf{w})$ is the negative log-likelihood function, as defined in the lecture (Eq. 33)

5.2 Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is 1. Run all the cells of the notebook. 2. Download the notebook in HTML (click File > Download as > .html) 3. Convert the HTML to PDF using e.g. https://www.sejda.com/html-to-pdf or `wkhtmltopdf` for Linux (tutorial) 4. Concatenate your solutions for other tasks with the output of Step 3. On a Linux machine you can simply use `pdfunite`, there are similar tools for other platforms too. You can only upload a single PDF file to Moodle.

This way is preferred to using `nbconvert`, since `nbconvert` clips lines that exceed page width and makes your code harder to grade.

5.3 Load and preprocess the data

In this assignment we will work with the UCI ML Breast Cancer Wisconsin (Diagnostic) dataset https://goo.gl/U2Uwz2.

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. There are 212 malignant examples and 357 benign examples.

```
In [4]: X, y = load_breast_cancer(return_X_y=True)

        # Add a vector of ones to the data matrix to absorb the bias term
        X = np.hstack([np.ones([X.shape[0], 1]), X])

        # Set the random seed so that we have reproducible experiments
        np.random.seed(123)

        # Split into train and test
        test_size = 0.3
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)
```

5.4 Task 1: Implement the sigmoid function

```
In [5]: def sigmoid(t):
            """
            Applies the sigmoid function elementwise to the input data.

            Parameters
            ----------
            t : array, arbitrary shape
                Input data.

            Returns
            -------
```

```
    t_sigmoid : array, arbitrary shape.
        Data after applying the sigmoid function.
    """
    return 1 / (1 + np.exp(-t))
```

5.5 Task 2: Implement the negative log likelihood

As defined in Eq. 33

```
In [17]: def negative_log_likelihood(X, y, w):
             """
             Negative Log Likelihood of the Logistic Regression.

             Parameters
             ----------
             X : array, shape [N, D]
                 (Augmented) feature matrix.
             y : array, shape [N]
                 Classification targets.
             w : array, shape [D]
                 Regression coefficients (w[0] is the bias term).

             Returns
             -------
             nll : float
                 The negative log likelihood.
             """
             return  -(y.dot(np.log(sigmoid(w.dot(X.T))).T)
                       + (1 - y).dot(np.log((1 - sigmoid(w.dot(X.T)))).T))
```

5.5.1 Computing the loss function $\mathcal{L}(\mathbf{w})$ (nothing to do here)

```
In [47]: def compute_loss(X, y, w, lmbda):
             """
             Negative Log Likelihood of the Logistic Regression.

             Parameters
             ----------
             X : array, shape [N, D]
                 (Augmented) feature matrix.
             y : array, shape [N]
                 Classification targets.
             w : array, shape [D]
                 Regression coefficients (w[0] is the bias term).
             lmbda : float
                 L2 regularization strength.

             Returns
             -------
             loss : float
                 Loss of the regularized logistic regression model.
             """
             # The bias term w[0] is not regularized by convention
             return negative_log_likelihood(X, y, w) / len(y) \
                    + lmbda * np.linalg.norm(w[1:])**2
```

5.6 Task 3: Implement the gradient $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$

Make sure that you compute the gradient of the loss function $\mathcal{L}(\mathbf{w})$ (not simply the NLL!)

```
In [39]: def get_gradient(X, y, w, mini_batch_indices, lmbda):
             """
             Calculates the gradient (full or mini-batch) of the negative log likelilhood w.r.t. w.

             Parameters
             ----------
             X : array, shape [N, D]
                 (Augmented) feature matrix.
             y : array, shape [N]
                 Classification targets.
             w : array, shape [D]
                 Regression coefficients (w[0] is the bias term).
             mini_batch_indices: array, shape [mini_batch_size]
                 The indices of the data points to be included in the (stochastic) calculation of the gr
                 This includes the full batch gradient as well, if mini_batch_indices = np.arange(n_tra
             lmbda: float
                 Regularization strentgh. lmbda = 0 means having no regularization.

             Returns
             -------
             dw : array, shape [D]
                 Gradient w.r.t. w.
             """

             X = X[mini_batch_indices]
             y = y[mini_batch_indices]
             return (1/len(X)) * X.T.dot(sigmoid(w.dot(X.T)) - y) - lmbda * w
```

5.6.1 Train the logistic regression model (nothing to do here)

```
In [40]: def logistic_regression(X, y, num_steps, learning_rate, mini_batch_size, lmbda, verbose):
             """
             Performs logistic regression with (stochastic) gradient descent.

             Parameters
             ----------
             X : array, shape [N, D]
                 (Augmented) feature matrix.
             y : array, shape [N]
                 Classification targets.
             num_steps : int
                 Number of steps of gradient descent to perform.
             learning_rate: float
                 The learning rate to use when updating the parameters w.
             mini_batch_size: int
                 The number of examples in each mini-batch.
                 If mini_batch_size=n_train we perform full batch gradient descent.
             lmbda: float
                 Regularization strentgh. lmbda = 0 means having no regularization.
             verbose : bool
                 Whether to print the loss during optimization.
```

```
    Returns
    -------
    w : array, shape [D]
        Optimal regression coefficients (w[0] is the bias term).
    trace: list
        Trace of the loss function after each step of gradient descent.
    """

    trace = [] # saves the value of loss every 50 iterations to be able to plot it later
    n_train = X.shape[0] # number of training instances

    w = np.zeros(X.shape[1]) # initialize the parameters to zeros

    # run gradient descent for a given number of steps
    for step in range(num_steps):
        permuted_idx = np.random.permutation(n_train) # shuffle the data

        # go over each mini-batch and update the paramters
        # if mini_batch_size = n_train we perform full batch GD and this loop runs only once
        for idx in range(0, n_train, mini_batch_size):
            # get the random indices to be included in the mini batch
            mini_batch_indices = permuted_idx[idx:idx+mini_batch_size]
            gradient = get_gradient(X, y, w, mini_batch_indices, lmbda)

            # update the parameters
            w = w - learning_rate * gradient

        # calculate and save the current loss value every 50 iterations
        if step % 50 == 0:
            loss = compute_loss(X, y, w, lmbda)
            trace.append(loss)
            # print loss to monitor the progress
            if verbose:
                print('Step {0}, loss = {1:.4f}'.format(step, loss))
    return w, trace
```

5.7 Task 4: Implement the function to obtain the predictions

```
In [41]: def predict(X, w):
    """
    Parameters
    ----------
    X : array, shape [N_test, D]
        (Augmented) feature matrix.
    w : array, shape [D]
        Regression coefficients (w[0] is the bias term).

    Returns
    -------
    y_pred : array, shape [N_test]
        A binary array of predictions.
    """
    return (X.dot(w.T) > 0)*1
```

5.7.1 Full batch gradient descent

```
In [10]: # Change this to True if you want to see loss values over iterations.
         verbose = False
```

```
In [42]: n_train = X_train.shape[0]
         w_full, trace_full = logistic_regression(X_train,
                                                  y_train,
                                                  num_steps=8000,
                                                  learning_rate=1e-5,
                                                  mini_batch_size=n_train,
                                                  lmbda=0.1,
                                                  verbose=verbose)
```

```
In [43]: n_train = X_train.shape[0]
         w_minibatch, trace_minibatch = logistic_regression(X_train,
                                                            y_train,
                                                            num_steps=8000,
                                                            learning_rate=1e-5,
                                                            mini_batch_size=50,
                                                            lmbda=0.1,
                                                            verbose=verbose)
```

Our reference solution produces, but don't worry if yours is not exactly the same.

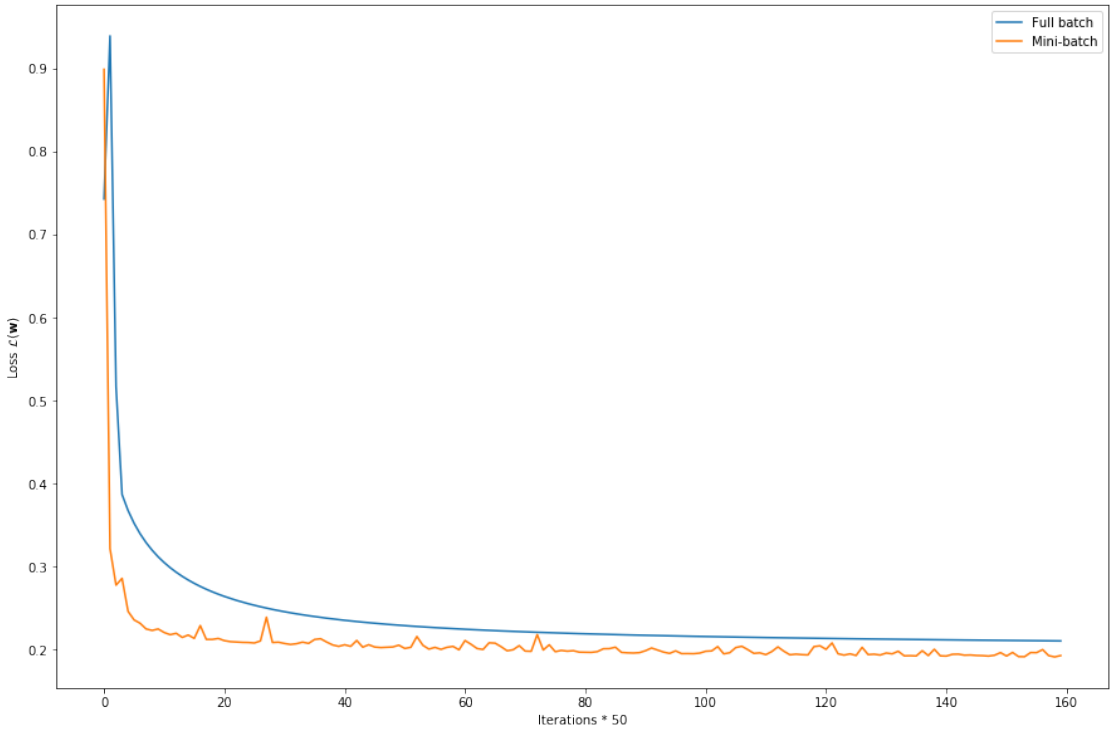
```
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9533
```

```
In [45]: y_pred_full = predict(X_test, w_full)
         y_pred_minibatch = predict(X_test, w_minibatch)

         print('Full batch: accuracy: {:.4f}, f1_score: {:.4f}'
               .format(accuracy_score(y_test, y_pred_full), f1_score(y_test, y_pred_full)))
         print('Mini-batch: accuracy: {:.4f}, f1_score: {:.4f}'
               .format(accuracy_score(y_test, y_pred_minibatch), f1_score(y_test, y_pred_minibatch)))
```

```
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9528
```

```
In [46]: plt.figure(figsize=[15, 10])
         plt.plot(trace_full, label='Full batch')
         plt.plot(trace_minibatch, label='Mini-batch')
         plt.xlabel('Iterations * 50')
         plt.ylabel('Loss $\mathcal{L}(\mathbf{w})$')
         plt.legend()
         plt.show()
```


Full batch
Mini-batch
Loss $\mathcal{L}(\mathbf{w})$
Iterations * 50
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
20
40
60
80
100
120
140
160