Machine learning Exercise sheet 6
Optimization

Anastasia Stamatouli : 03710902

24/11/19

Convexity of functions

Problem ① :

a) $h(x) = g_2(g_1(x))$

$h'(x) = g_2'[g_1(x)] \cdot g_1'(x)$

$h''(x) = \underset{\geq 0}{g_2''[g_1(x)]}\,\underset{\geq 0}{[g_1'(x)]^2} + \underset{?}{g_2'[g_1(x)]} \cdot \underset{\geq 0}{g''(x)}$

we know that $g_1(x) \to$ convex and $g_2(x) \to$ convex

then : $g_1''(x) \geq 0$

$g_2''(x) \geq 0$

$[g_1'(x)]^2 \geq 0$

In order $h(x)$ to be convex, then $h''(x) \geq 0$, but we don't know what $g_2'[g_1(x)]$ is. It can be both positive and negative. So the function is not always convex.

b) $h(x) = g_2(g_1(x))$

$h'(x) = g_2'[g_1(x)] \cdot g_1'(x)$

$h''(x) = g_2''\underset{\geq 0}{[g_1(x)]} \cdot \underset{\geq 0}{[g_1'(x)]^2} + g_2'\underset{\geq 0}{[g_1(x)]} \cdot \underset{\geq 0}{g''(x)}$

g_1, g_2 convex $\to$ $(g_2(x))'' \geq 0$

$g_1''(x) \geq 0$

g_2 non decreasing $\to g_2'(x) \geq 0$

Therefore $h(x)$ convex.

c) Pick any $x, y \in dom(h)$, $\lambda \in [0,1]$.

Then,

$$h(\lambda x + (1-\lambda)y) = g_j(\lambda x + (1-\lambda)y)$$

for some $j \in \{1, \cdots, n\}$

$$\overset{g_j \text{ convex}}{\leq} \lambda g_j(x) + (1-\lambda) g_j(y)$$

$$\leq \lambda \max(g_1, \cdots, g_n(x)) + (1-\lambda)\max(g_1(y) \cdots g_j(y))$$

$$= \lambda h(x) + (1-\lambda) f(y) \quad \square$$

- Optimization / Gradient Descent

Problem ②:

a) $\frac{df}{dx_1} = x_1 + 2$, $\frac{df}{dx_2} = 2x_2 + 1$

$$\nabla f(x_1, x_2) = \begin{bmatrix} x_1 + 2 \\ 2x_2 + 1 \end{bmatrix}$$

$$\nabla f = 0 \iff \boxed{x_1 = -2} \quad \boxed{x_2 = -\frac{1}{2}}$$

$$Hf(x_1, x_2) = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

To determine, whether a matrix is positive definite, we first compute it's determinant.

$$\det(H) = 2 - 0 = 2 > 0 \quad (1)$$

The product of the eigenvalues is positive, which means they have same sign.

We check the $tr(H)$ which is the sum of eigenvalues in order to determine the sign.

$tr(H) = 1 + 2 = 3$ (2) $\overset{(1),(2)}{\Longrightarrow}$ both eigenvalues positive

Therefore $H \to$ positive definite and $(x_1, x_2) = (-2, -\frac{1}{2})$ local minimum.

b) $\begin{pmatrix} X_1 n \\ X_2 n \end{pmatrix} = \begin{pmatrix} X_1 n-1 \\ X_2 n-1 \end{pmatrix} - \alpha \nabla f \left(X_1 n-1 , X_2 n-1 \right)$ 2

our original function value is :

$$f(0,0) = \cos(\sin(\sqrt{\pi})) = 0.9999..$$

$$\nabla f(X_1, X_2) = \begin{bmatrix} X_1 + 2 \\ 2X_2 + 1 \end{bmatrix}$$

$$\nabla f(0,0) = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$\boxed{\text{Step 1}}$

$$\begin{pmatrix} X_1 n \\ X_2 n \end{pmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix}$$

$$f(X_1 n, X_2 n) = -1,000000146$$

$\boxed{\text{Step 2}}$

$$\nabla f(X_1 n, X_2 n) = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

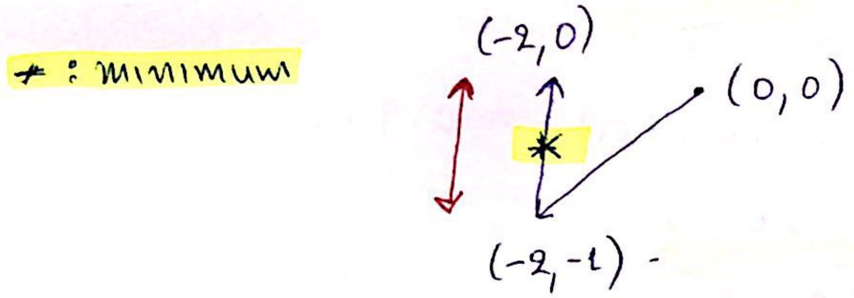
$$\begin{bmatrix} X_1 n \\ X_2 n \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} -2 \\ 0 \end{bmatrix}$$

$$f(X_1 n, X_2 n) = -1,000000146$$

minimum value for $(X_1, X_2) = (-2, \frac{-1}{2}) \Rightarrow$

$$f(X_1, X_2) = -1.75000146$$

c) if we draw the isolines. we see the. following behaviour :



Gradient descent will not converge to the minimum x*. As we see above, we overshoot and then oscilate between the points (-2,-1) and (-2, 0). In order to overcome this problem we need smaller step, which means smaller learning rate. We could also have adaptive learning rate where at the beginning we do larger steps and later on we do smaller in order not to overshoot the minimum.

Problem 4 :

a) The shaded region is not convex because we cannot pass between the points (3.5, 1) and (6, 3.5) without leaving the shaded region.

b) In order to find maximizer x^* of of over the shaded region S we will split the non-convex set into smaller convex ones which are the following ones.

4 triangles

1 square in the middle.

Afterwards, we will find the maximum value in each one of the convex sets and then pick the largest one.

We start working on the right triangle:

Maximum over a convex function on a convex set is obtained on a vertex.

Therefore, we have to check the vertexes in the right triangle. The vertexes are the following

- (6, 3.5)
- (4.5, 3)
- (4.5, 4)

$$f(6, 3.5) = e^{6+3.5} - 5\log(3.5) = 13357{,}00649$$

$$f(4.5, 3) = e^{4.5+3} - 5\log(3) = 1805{,}0322114$$

$$f(4.5, 4) = e^{4.5+4} - 5\log(4) = 4911{,}75854$$

We do the same process for the rest convex sets and we choose the maximum value out of all of them.

c) in order to find the minimum we can follow the same process as before but instead we will transform the function to concave $(-f(x_1,x$ and find minimum in the vertices.

exercise_06_optimization

November 24, 2019

1 Programming assignment 3: Optimization - Logistic Regression

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%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
```

1.1 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 (see Eq. 33).

1.2 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. Export/download the notebook as PDF (File -> Download as -> PDF via LaTeX (.pdf)). 3. Concatenate your solutions for other tasks with the output of Step 2. 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.

Make sure you are using `nbconvert` Version 5.5 or later by running `jupyter nbconvert --version`. Older versions clip lines that exceed page width, which makes your code harder to grade.

1.3 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.

[2]:
```
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)
```

1.4 Task 1: Implement the sigmoid function

[4]:
```
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.
    """
    t_sigmoid = 1/ (1+np.exp(-t))
    return t_sigmoid
```

1.5 Task 2: Implement the negative log likelihood

As defined in Eq. 33

[7]:
```
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.
    """
    nll = -(y.dot(np.log(sigmoid(w.dot(X.T))).T)
              + (1 - y).dot(np.log((1 - sigmoid(w.dot(X.T)))).T))
    return nll
```

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

[8]:
```
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 * 0.5 * np.linalg.
 ↪norm(w[1:])**2
```

1.6 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!)

[9]:
```
def get_gradient(X, y, w, mini_batch_indices, lmbda):
    """
    Calculates the gradient (full or mini-batch) of the negative log likelihood␣
 ↪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 gradient.
        This includes the full batch gradient as well, if mini_batch_indices =␣
 ↪np.arange(n_train).
    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
```

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

[10]:
```
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
```

1.7 Task 4: Implement the function to obtain the predictions

[11]:
```
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
```

1.7.1 Full batch gradient descent

[12]:
```
# Change this to True if you want to see loss values over iterations.
verbose = False
```

[13]:
```
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)
```

[14]:
```
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.

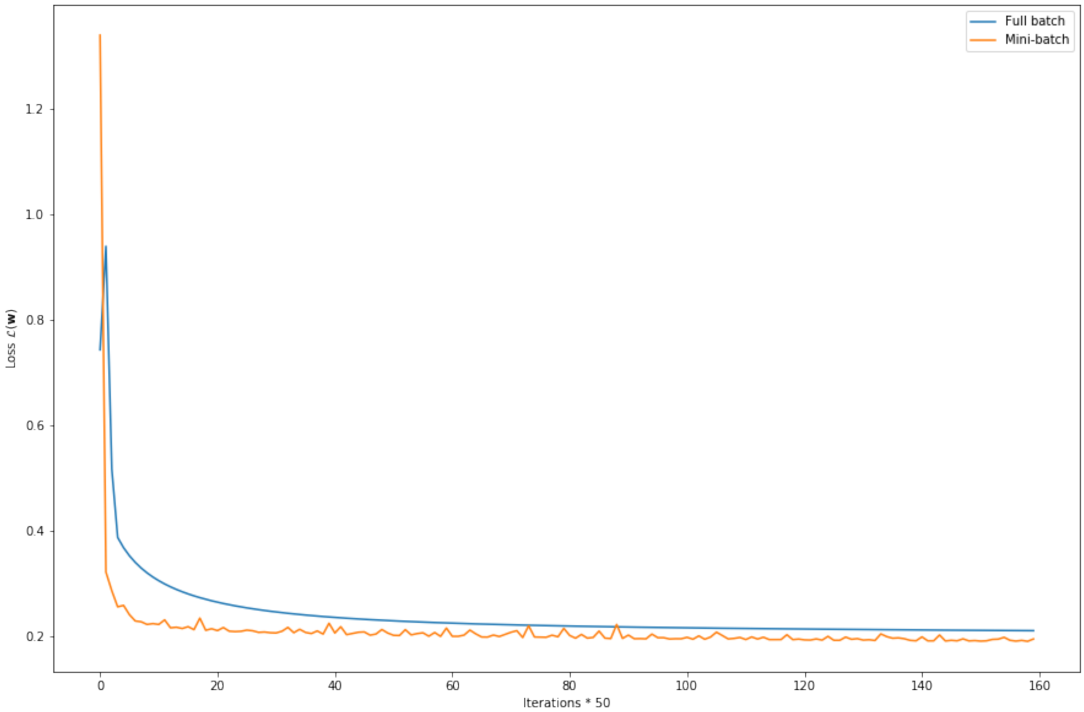
```
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9533
```

[15]:
```python
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: 0.9533
```

[16]:
```python
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()
```



[ ]:
```python

```