## Machine Learning Exercise Sheet 6

# Optimization

**Group_369**

Fan Xue – fan98.xue@tum.de

Xing Zhou – xing.zhou@tum.de

Jianzhe Liu – jianzhe.liu@tum.de

December 1, 2021

---

### Problem 4

a) This statement is false, which can be proved by a counterexample. For that, we need to adopt an inference that the second derivation of a convex function is non-negative. We take $f(x) = x^2$ and $g(x) = -x$, both of which are convex funtions, since $\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = 2 \geq 0$ and $\frac{\mathrm{d}^2 g(x)}{\mathrm{d}x^2} = 0 \geq 0$. Then we take the composition of the two functions as $h(x) = g(f(x))$, and we have:

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} h(x) = -2 \leq 0.$$

That means $h(x) = g(f(x))$ is non-convex.

b) This statement is true. We prove it with the definition of convex function. We take $h(x) = g(f(x))$, $\forall x_1, x_2 \in \mathbb{R}$ and $t \in (0, 1)$, since $f(x)$ is convex we have

$$f\left(tx_1 + (1-t)x_2\right) \leq tf\left(x_1\right) + (1-t)f\left(x_2\right).$$

Since $g(x)$ is non-decreasing and convex, we have

$$g\left(f\left(tx_1 + (1-t)x_2\right)\right) \leq g\left(tf\left(x_1\right) + (1-t)f\left(x_2\right)\right) \tag{1}$$

$$g\left(tf(x_1) + (1-t)f(x_2)\right) \leq tg\left(f(x_1)\right) + (1-t)g\left(f(x_2)\right). \tag{2}$$

Combine (1) and (2), we have

$$\begin{gathered} g\left(f\left(tx_1 + (1-t)x_2\right)\right) \leq tg\left(f(x_1)\right) + (1-t)g\left(f(x_2)\right) \\ \Updownarrow \\ h\left(tx_1 + (1-t)x_2\right) \leq th(x_1) + (1-t)h(x_2), \end{gathered}$$

which proves $h(x) = g(f(x))$ is convex.

## Problem 5

a) By observing the given function $\boldsymbol{f}$:

$$f(x_1, x_2) = 0.5{x_1}^2 + {x_2}^2 + 2x_1 + x_2 + \cos(\sin(\sqrt{\pi}))$$

we can find that it consists of several convex functions. Which means that when we separately find all the $x_n$ that minimize its own function, we find the $\boldsymbol{x}^\star$ that minimizes the function $\boldsymbol{f}$.
The gradient of f can be computed by followings:

$$\begin{pmatrix} \dfrac{\partial f(x_1, x_2)}{\partial x_1} \\ \dfrac{\partial f(x_1, x_2)}{\partial x_2} \end{pmatrix} = \begin{pmatrix} x_1 + 2 \\ 2x_2 + 1 \end{pmatrix} = 0$$

So minimizer $\boldsymbol{x}^\star$ is:

$$\boldsymbol{x}^\star = \begin{pmatrix} -2 \\ -\dfrac{1}{2} \end{pmatrix}$$

b) 2 steps of gradient descent with leaning rate $\tau = 1$ can be performed by following procedure:

$$\text{step 1: } \begin{pmatrix} {x_1}^{(1)} \\ {x_2}^{(1)} \end{pmatrix} = \begin{pmatrix} {x_1}^{(0)} \\ {x_2}^{(0)} \end{pmatrix} - \tau \nabla_{\boldsymbol{x}} f\left(\boldsymbol{x}^{(0)}\right) = \begin{pmatrix} -2 \\ -1 \end{pmatrix}$$

$$\text{step 2: } \begin{pmatrix} {x_1}^{(2)} \\ {x_2}^{(2)} \end{pmatrix} = \begin{pmatrix} {x_1}^{(1)} \\ {x_2}^{(1)} \end{pmatrix} - \tau \nabla_{\boldsymbol{x}} f\left(\boldsymbol{x}^{(1)}\right) = \begin{pmatrix} -2 \\ 0 \end{pmatrix}$$

Here $\nabla f\left(\boldsymbol{x}\right)$ is obtained from **question a)** above, where

$$\frac{\partial f(x_1, x_2)}{\partial x_1} = x_1 + 2$$

$$\frac{\partial f(x_1, x_2)}{\partial x_2} = 2x_2 + 1$$

c) we can try computing two more iteration:

$$\begin{pmatrix} {x_1}^{(3)} \\ {x_2}^{(3)} \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \end{pmatrix} - 1 \begin{pmatrix} -2 + 2 \\ 0 + 1 \end{pmatrix} = \begin{pmatrix} -2 \\ -1 \end{pmatrix}$$

$$\begin{pmatrix} {x_1}^{(4)} \\ {x_2}^{(4)} \end{pmatrix} = \begin{pmatrix} -2 \\ -1 \end{pmatrix} - 1 \begin{pmatrix} -2 + 2 \\ -2 + 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \end{pmatrix}$$

Obviously, it's an endless loop.....Yet we already know that minimizer $\boldsymbol{x}^\star$ is: $\begin{pmatrix} -2 \\ \frac{1}{2} \end{pmatrix}$.

So the answer is, this gradient descent procedure can never converge to the true minimizer $\boldsymbol{x}^\star$.
To solve this problem, we must change the learning rate. In our case, it has to decrease to get to the minimizer.

## Problem 6

The result of the programming task is appended at the end of the file.

## Problem 7

a) This region S is not a convex region. Because we can find points whose line in between is outside region S. As shown in figure 1, red, green and blue lines are all in this situation.

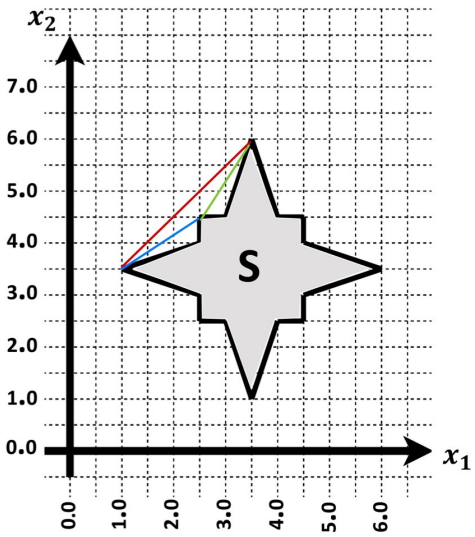



Figure 1: Figure 1

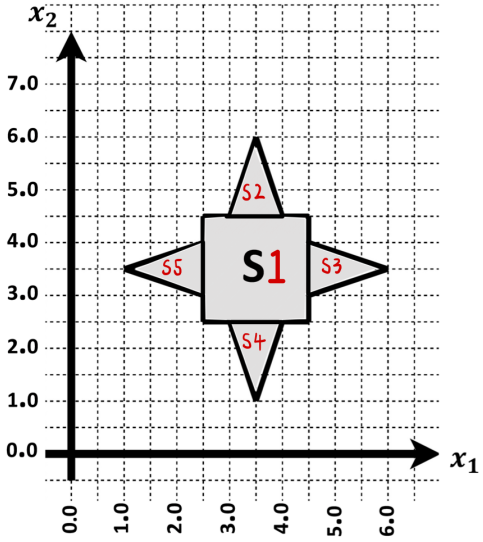



Figure 2: Figure 2

b) It's obvious that Function F is meaningful only if it's region is convex. So to give this **ConvOpt** algorithm a qualified input, we need to divide the original region S into 5 small regions, which are all convex regions(as shown in figure2).

This way, we can use this **ConvOpt** algorithm to get 5 different results, which are respectively the minimum in their region. Then we choose the minimum among this 5 values, it is the minimum of whole region S.

# exercise_06_notebook

December 1, 2021

## 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

[3]:
```
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.
    """
    # TODO
    t_sigmoid = 1 / (1 + np.exp(-t))
    return t_sigmoid
```

### 1.5 Task 2: Implement the negative log likelihood

As defined in Eq. 33

[4]:
```
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.
    """
    # TODO
    sigmoid_Xw = sigmoid(np.dot(X, w))
    nll = -(np.dot(y, np.log(sigmoid_Xw)) + np.dot(1 - y, np.log(1 -␣
↪sigmoid_Xw)))
    return nll
```

[5]:
```
# y = np.array([1, 2, 3])
# s = np.array([[1],
#               [2],
#                [3]])
# print(1-y)
# print(np.dot(y,s))
# print(y.T)
# print(y.reshape(3, 1))
```

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

[6]:
```
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!)

[7]:
```
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 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.
    """
    # TODO
    # N = y.shape[0]
    # print(N)
    X = X[mini_batch_indices, ]
    y = y[mini_batch_indices]
    N_new = len(mini_batch_indices)
    dw = (-1 / N_new) * np.dot(X.T, y - sigmoid(np.dot(X, w))) + lmbda * w
    return dw
```

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

[8]:
```
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

[39]:
```
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.
    """
    # TODO
    y_pred = (sigmoid(np.dot(X, w)))
    # y_pred = y_pred > 0.5
    y_pred[y_pred > 0.5] = int(1)
    y_pred[y_pred <= 0.5] = int(0)
    print(y_pred)
    return y_pred
```

#### 1.7.1 Full batch gradient descent

[40]:
```
# Change this to True if you want to see loss values over iterations.
verbose = False
```

[41]:
```
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)
```

[42]:
```
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
```

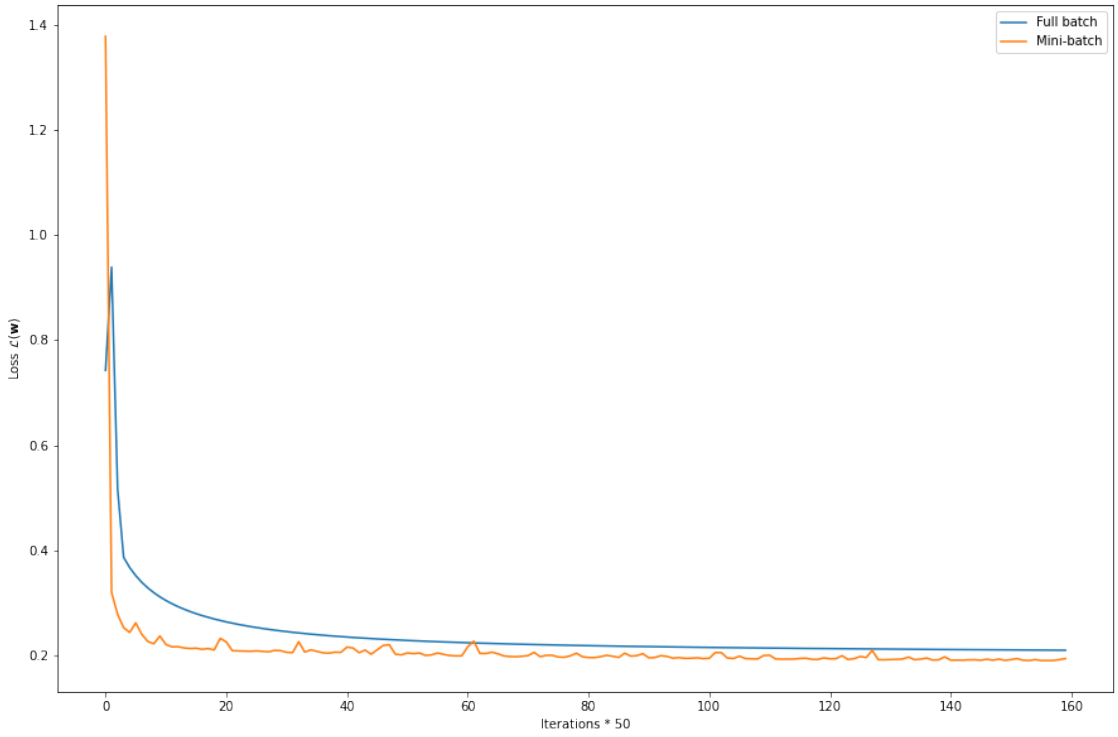
[43]:
```
y_pred_full = predict(X_test, w_full)
y_pred_minibatch = predict(X_test, w_minibatch)
print("y_test:", y_test)

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)))
```

```
[1. 1. 0. 1. 0. 1. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1.
 1. 0. 0. 1. 0. 1. 0. 1. 1. 1. 0. 1. 0. 1. 1. 0. 0. 1. 0. 1. 0. 1. 0. 0.
 1. 0. 0. 0. 1. 1. 1. 1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1.
 0. 1. 0. 1. 1. 0. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1. 0. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1.
 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 1. 0. 1.
 0. 1. 0. 0. 1. 0. 1. 0. 1. 1. 1. 1. 0. 0. 0. 1. 0. 1. 0. 0. 1. 1. 1. 0.
 0. 0. 1.]
[1. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1.
 1. 0. 0. 1. 0. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 0. 0. 1. 0. 1. 0. 1. 0. 0.
 1. 0. 0. 0. 1. 1. 1. 1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1.
 0. 1. 0. 1. 1. 0. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1. 0. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1.
 1. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 1. 0. 1.
 0. 1. 0. 0. 1. 0. 1. 0. 1. 1. 1. 1. 0. 0. 0. 1. 0. 1. 0. 0. 1. 1. 1. 0.
 0. 0. 1.]
y_test: [1 1 0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 1 1 1 0 0 1 1 1 0 0 1 0 1 0 1 1 1 0 1
```

```
1
 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 1 1 0 1 0 0 1 0 1 1 1 1 0 1 1 1 0 1 1 0 1
 0 1 1 0 0 0 1 0 0 1 1 1 0 1 0 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 0 0 0 1 0 1 1 1 1 1 1 0 1 0 0 1 1 0 1 1 0 0 1 1 1 0 0 0 1 0 1 0 1 0 0
 1 0 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 1 1 0 0 0 0]
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9528
```

[32]:
```
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()
```




[ ]: