Machine Learning: Homework #6

Due on December 04, 2017 at 09:59am

Professor Dr. Stephan Guennemann

Marc Meissner - 03691673

Problem 1

Prove or disprove whether the following functions are convex on the given set D:

i) $f(x, y, z) = 3x + e^{y+z} - \min(-x^2, \log(y))$ and $D = (-100, 100) \times (1, 50) \times (10, 20)$
ii) $f(x, y) = yx^3 - 2yx^2 + y + 4$ and $D = (-10, 10) \times (-10, 10)$
iii) $f(x) = log(x) + x^3$ and $D = (1, \infty)$
iv) $f(x) = -\min(2\log(2x), -x^2 + 4x - 32)$ and $D = \mathbb{R}^+$

Solution
i)
One can immediately see that $\min(-x^2, \log(y)) = -x^2$ for the given set D. Ergo:
$f(x, y, z) = 3x + e^{y+z} - \min(-x^2, \log(y)) = 3x + e^{y+z} + x^2$
Since these are just additions of convex functions, f is convex.

ii)
Let $\vec{x_1} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$, $\vec{x_2} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $\lambda = \frac{1}{2}$. Then

$\lambda f(\vec{x_1}) + (1 - \lambda) f(\vec{x_2}) \geq f(\lambda \vec{x_1} + (1 - \lambda)\vec{x_2})$

would have to hold for the function to be convex. But

$\lambda f(\vec{x_1}) + (1 - \lambda) f(\vec{x_2}) = 3$ and $f(\lambda \vec{x_1} + (1 - \lambda)\vec{x_2}) = 4.1875$

Here, $\lambda f(\vec{x_1}) + (1 - \lambda) f(\vec{x_2}) < f(\lambda \vec{x_1} + (1 - \lambda)\vec{x_2})$ and thus the function is not convex.

iii)
If the second derivative is non-negative for the given set D, it is convex. Calculating the derivatives and checking the second one proves the convexity of f:

$\frac{df}{dx} = \frac{1}{x} + 3x^2$ and $\frac{d^2f}{dx^2} = -\frac{1}{x^2} + 6x > 0$ for $x \in [1; \infty]$

iv)
Intuitively, the log function goes to $-\infty$ for small x values:

$lim_{x \to 0} - \min(2\log(2x), -x^2 + 4x - 32) = -\min(-\infty, -32) = \infty$

The parabola is convex anyway. At one point, the log function becomes smaller than the parabola. Since the negative log is convex for $0 < x < 1$, f is convex as well.

Problem 2

Prove the following statement: Let $f_1 : \mathbb{R}^d \to \mathbb{R}$ and $f_2 : \mathbb{R}^d \to \mathbb{R}$ be convex functions, then $h(x) = f_1(x) + f_2(x)$ is also a convex function.

Solution
Using the definition of convexity:

$\lambda h(x) + (1-\lambda)h(y)$
$= \lambda(f_1(x) + f_2(x)) + (1-\lambda)(f_1(y) + f_2(y))$
$= \lambda f_1(x) + (1-\lambda)f_1(y) + \lambda f_2(x) + (1-\lambda)f_2(y)$
$\geq f_1(\lambda x + (1-\lambda)y) + f_2(\lambda x + (1-\lambda)y)$
$= h(\lambda x + (1-\lambda)y)$

q.e.d. bish

Problem 3

Given two convex functions $f_1 : \mathbb{R} \to \mathbb{R}$ and $f_2 : \mathbb{R} \to \mathbb{R}$, prove or disprove that the function $h(x) = f_1(x) \cdot f_2(x)$ is also convex.

Solution
Simple counterexample: imagine two linear (and thus convex) functions $f_1(x) = x$ and $f_2(x) = -x$. The product $g(x) = x \cdot (-x) = -x^2$ is obviously not convex.

Problem 4

Prove that for convex functions each local minimum is a global minimum. More specifically, given a convex function $f : \mathbb{R}^N \to \mathbb{R}$, prove that if $\nabla f(\theta^*) = 0$ then θ^* is a global minimum.

Solution

From the definition of first order convexity:

$f(y) \geq (y-x)^T \nabla f(x) + f(x)$

Setting $x = x^*$ and abusing $\nabla f(x^*) = 0$, we proof the theorem:

$f(y) \geq (y-x^*)^T \nabla f(x^*) + f(x^*) = f(x^*)$
$f(y) \geq f(x^*)$

Problem 5

Load the notebook 06_*hw_optimization_logistic_regression.ipynb* from Piazza. Fill in the missing code and run the notebook. Convert the evaluated notebook to pdf and add it to the printout of your homework.

Solution

The notebook pdf is added.

06_hw_optimization_logistic_regression_v2

December 1, 2017

1 Programming assignment 6: Optimization: Logistic regression

```
In [2]: 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 (Eq. 33)

1.2 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 [3]: 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)
```

1.3 Task 1: Implement the sigmoid function

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

1.4 Task 2: Implement the negative log likelihood

As defined in Eq. 33

```
In [5]: 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
            sig = sigmoid(np.matmul(X, w))
```

```
    nll = -np.sum(y * np.log(sig+1e-15) + (1 - y) * np.log(1 - sig+1e-15))
    return nll
```

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

```
In [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 * np.linalg.norm(w[1:])**2
```

1.5 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 [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.

            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
                This includes the full batch gradient as well, if mini_batch_indices = np.arange
            lmbda: float
```

```
        Regularization strentgh. lmbda = 0 means having no regularization.

    Returns
    -------
    dw : array, shape [D]
        Gradient w.r.t. w.
    """
    # TODO
    N_batch = len(mini_batch_indices)
    D = len(w)
    dw = np.zeros(D)
    for j in range(D):
        dw[j] = 1 / N_batch * \
                np.matmul((sigmoid(np.matmul(X[mini_batch_indices], w)) - y[mini_batch_i
                          , X[mini_batch_indices, j]) \
                + lmbda * w[j]
    return dw
```

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

```
In [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
        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
```

1.6 Task 4: Implement the function to obtain the predictions

```
In [9]: 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 = np.matmul(X,w)
```

```
        y_pred[y_pred<0] = 0
        y_pred[y_pred>0] = 1
        return y_pred
```

1.6.1 Full batch gradient descent

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

```
In [11]: 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 [12]: 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.

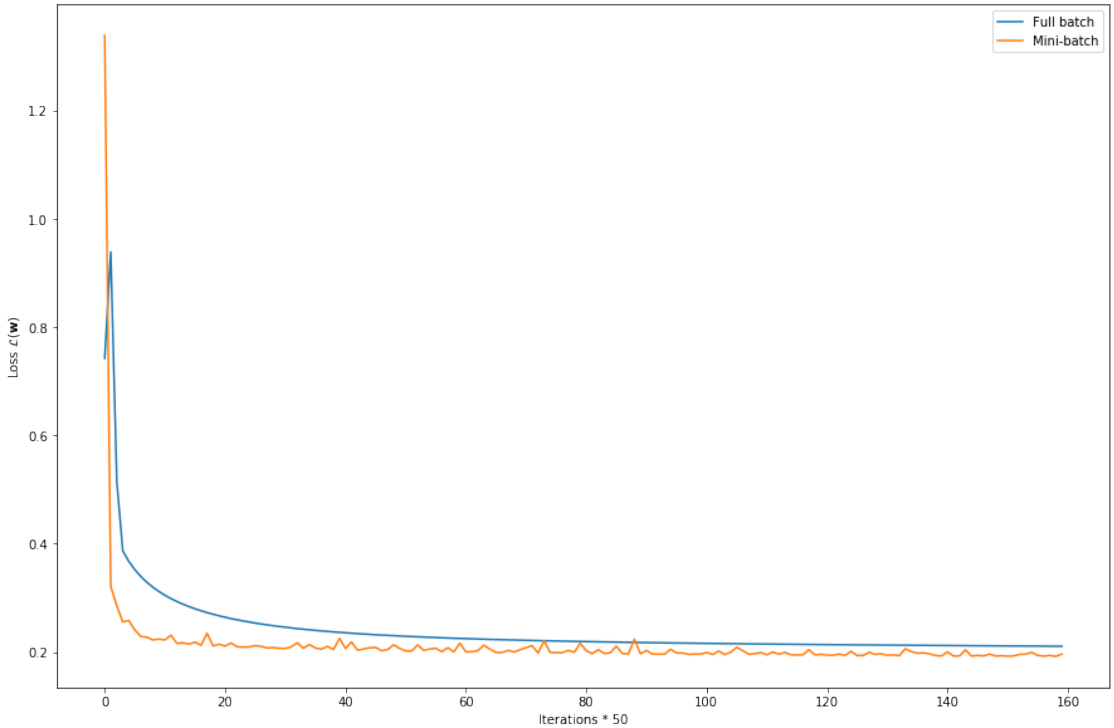
```
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9533
```

```
In [13]: 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_minibat
```

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

```
In [14]: 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
1.2
1.0
0.8
0.6
0.4
0.2
Loss $\mathcal{L}(\mathbf{w})$
0
20
40
60
80
100
120
140
160
Iterations * 50