Exercise

01

TUM Department of Informatics

Supervised by Prof. Dr. Stephan Günnemann
Informatics 3 - Professorship of Data Mining and Analytics

Submitted by Marcel Bruckner (03674122)
Julian Hohenadel (03673879)
Kevin Bein (03707775)

Submission date Munich, November 24, 2019

Optimization

Problem 1:

a)

$h(x) = g_2(g_1(x))$ is not convex in this context:

Since g_2 and g_1 are convex: $g_2'' \geq 0$ and $g_1'' \geq 0$ but g_2 can be decreasing so $g_2'(x)$ does not have to be positive $\forall x$.

$$[h(x)]' = g_2'(g_1(x)) * g_1'(x)$$
$$[h(x)]'' = [g_2'(g_1(x)) * g_1'(x)]'$$
$$[h(x)]'' = g_2''(g_1(x)) * (g_1'(x))^2 + g_2'(g_1(x)) * g_1''(x)$$

So $[h(x)]''$ can be ≤ 0 if $g_2'(g_1(x)) \leq 0$ ($(g_1'(x))^2$ is always ≥ 0)

As an example: $g_2(x) = -\frac{1}{2}x$ and $g_1(x) = x^2$ are convex $\implies h(x) = -\frac{1}{2}x^2$ and should also be convex.

But the second derivation of $h(x)$ is -1 and therefore not convex.

b)

$h(x) = g_2(g_1(x))$ is convex in this context:

Since g_2 and g_1 are convex: $g_2'' \geq 0$ and $g_1'' \geq 0$ also g_2 is non-decreasing so $g_2'(x) \geq 0$.

$$[h(x)]' = g_2'(g_1(x)) * g_1'(x)$$
$$[h(x)]'' = [g_2'(g_1(x)) * g_1'(x)]'$$
$$[h(x)]'' = g_2''(g_1(x)) * (g_1'(x))^2 + g_2'(g_1(x)) * g_1''(x)$$

So $[h(x)]''$ can only be ≥ 0. ($(g_1'(x))^2$ is always ≥ 0)

c)

$h(x) = max(g_1(x), \dots, g_n(x))$ is always convex:

$$\begin{aligned} h(\lambda x + (1-\lambda)y) &= max(g_1(\lambda x + (1-\lambda)y), \dots, g_n(\lambda x + (1-\lambda)y) \\ &\leq max(\lambda g_1(x) + (1-\lambda)g_1(y), \dots, \lambda g_n(x) + (1-\lambda)g_n(y)) \\ &\leq max(\lambda g_1(x), \dots, \lambda g_n(x)) + max((1-\lambda)g_1(y), \dots, (1-\lambda)g_n(y)) \\ &= \lambda h(x) + (1-\lambda)h(y) \end{aligned}$$

Problem 2:

a)

Minimum $x*$ of f is the partial derivative wrt. x_1 and x_2:

$$\frac{\partial f}{\partial x_1} = x_1 + 2 \overset{!}{=} 0 \implies x_1 = -2$$
$$\frac{\partial f}{\partial x_2} = 2x_2 + 1 \overset{!}{=} 0 \implies x_2 = -\frac{1}{2}$$

$x*$ is at $x_1 = -2$ and $x_2 = -\frac{1}{2}$.

b)

2 steps gradient descent with $x^{(0)} = (0, 0)$ and learning rate $\tau = 1$:

Gradient descent in general:

1) Take point $x^{(n)}$

2) compute $f'(x)$

3) $x^{(n+1)} = x^{(n)} - \tau * f'(x)$

First step:

$$\frac{\partial f}{\partial x_1 = 0} = x_1 + 2 = 2 \implies x_1^{(1)} = 0 - 1 * 2 = -2$$
$$\frac{\partial f}{\partial x_2 = 0} = 2x_2 + 1 = 1 \implies x_2^{(1)} = 0 - 1 * 1 = -1$$
$$\implies x^{(1)} = (-2, -1)$$

Second step:

$$\frac{\partial f}{\partial x_1 = -2} = x_1 + 2 = 0 \implies x_1^{(2)} = -2 - 1 * 0 = -2$$
$$\frac{\partial f}{\partial x_2 = -1} = 2x_2 + 1 = -1 \implies x_2^{(2)} = -1 - 1 * -1 = 0$$
$$\implies x^{(2)} = (-2, 0)$$

c)

It will with $x_1 = -2$ but it won't with x_2 because it alternates between -1 and 0.

To solve this problem a learning rate $0 < \tau < 1$ would be needed to stop the alternation of x_2 and help to converge to $x_2 = -\frac{1}{2}$.

Problem 4:

a)

Regin S is not convex.

$$\forall x, y \in S : \lambda x + (1-\lambda)y \in S \qquad \forall \lambda \in [0,1]$$

Let x be $(3.5, 1) \in S$ and let y be $(6, 3.5) \in S$ and let λ be 0.5:

$$0.5 * (3.5, 1) + 0.5 * (6, 3.5) = (4.75, 2.25) \notin S$$

$\implies S$ is not convex

b)

Since f is a convex function, we know from the lecture that the maximum must lie on one of the region's vertices of the convex hull. The convex hull consists of $(3.5, 6.0), (6.0, 3.5), (3.5, 1.0), (1.0, 3.5)$ and does NOT contain the "smaller spikes" $(2.5, 4.5), (4.5, 4.5), (4.5, 2.5), (2.5, 2.5)$ We can simply calculate the values and find the maximum:

$$f(3.5, 6.0) = e^{9.5} - 5.0 \cdot \log 6.0 \approx 13350.8$$
$$f(6.0, 3.5) = e^{9.5} - 5.0 \cdot \log 3.5 \approx \mathbf{13353.5}$$
$$f(3.5, 1.0) = e^{4.5} - 5.0 \cdot \log 1.0 \approx 90.0$$
$$f(1.0, 3.5) = e^{4.5} - 5.0 \cdot \log 3.5 \approx 83.8$$

Therefore the maximum lies at $f(6.0, 3.5)$

c)

- Subdivide S into convex subregions.
- Use the algorithm to compute the minimum of every subdomain of S.
- Pick the one with the best minimum.

Problem 3:

exercise_06_optimization

November 24, 2019

1 Programming assignment 3: Optimization - Logistic Regression

```
In [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.

```
In [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

```
In [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.
            """
            return 1/(1+np.exp(-t))

        # Testing
        print(sigmoid(np.array([[1,2],[3,4]])))
```

```
[[0.73105858 0.88079708]
 [0.95257413 0.98201379]]
```

1.5 Task 2: Implement the negative log likelihood

As defined in Eq. 33

```
In [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.
        """
        scores = sigmoid(X.dot(w))
        nll = -np.sum(y*np.log(scores) + (1-y)*np.log(1-scores))
        return nll
```

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

```
In [5]: 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:])**
```

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!)

```
In [6]: 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
        This includes the full batch gradient as well, if mini_batch_indices = np.aran
    lmbda: float
        Regularization strentgh. lmbda = 0 means having no regularization.

    Returns
    -------
    dw : array, shape [D]
        Gradient w.r.t. w.
    """
    # https://math.stackexchange.com/questions/2503428/derivative-of-binary-cross-entr
    # dw = w - X^t(sigmoid(w^TX) - y) // This throws div by 0, but dw = X^t(sigmoid(w^
    # normalize: dw /= mini_batch_size
    # add regularization: dW += lmbda * W
    mini_batch_size = mini_batch_indices.shape[0]
    dW = X[mini_batch_indices].T.dot(sigmoid(X[mini_batch_indices].dot(w)) - y[mini_ba
    dW /= mini_batch_size
    dW += lmbda * w
    return dW
```

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

```
In [7]: def logistic_regression(X, y, num_steps, learning_rate, mini_batch_size, lmbda, verbos
    """
    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 lat
        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 onl
            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

```
In [8]: 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.
    """
    # this was painful
    # why can't just return np.argmax(sigmoid(X.dot(w))) or return (np.argmax(sigmoid(
    return (sigmoid(X.dot(w)) > 0.5).astype(np.int)
```

1.7.1 Full batch gradient descent

```
In [9]: # Change this to True if you want to see loss values over iterations.
        verbose = False
```

```
In [10]: 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 [11]: 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.

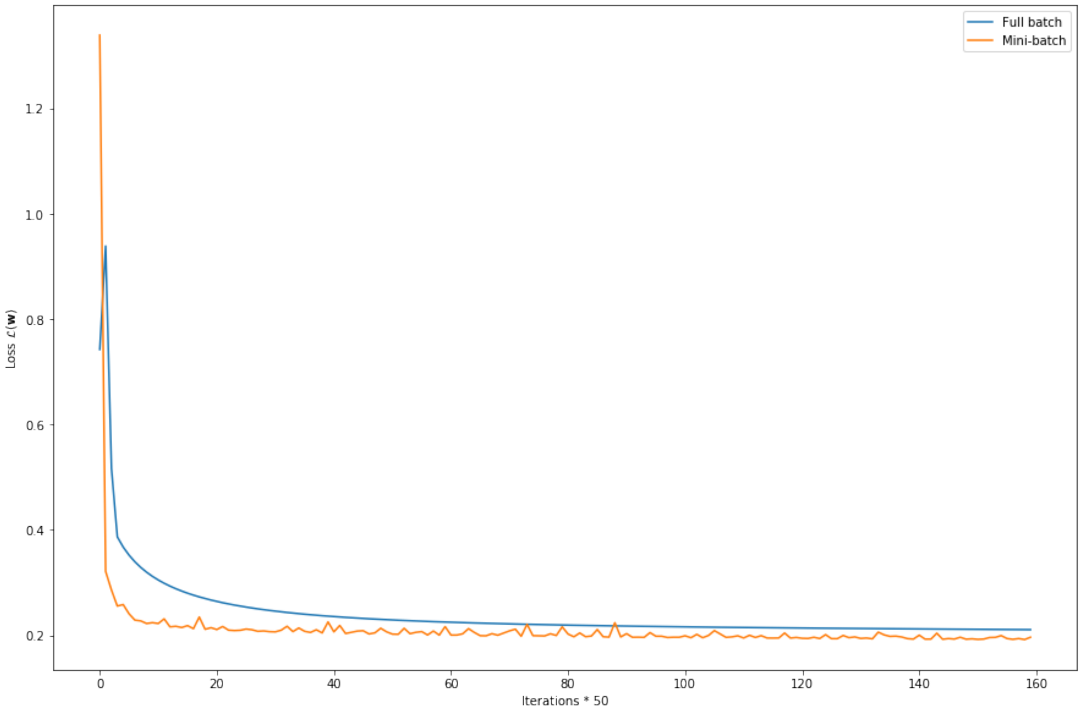
```
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9533
```

```
In [12]: 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_miniba
```

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

In [13]:
```
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()
```



In [ ]:

Appendix

We confirm that the submitted solution is original work and was written by us without further assistance. Appropriate credit has been given where reference has been made to the work of others.

Munich, November 24, 2019, Signature Marcel Bruckner (03674122)

Munich, November 24, 2019, Signature Julian Hohenadel (03673879)

Munich, November 24, 2019, Signature Kevin Bein (03707775)