Homework Assignment 6 - Machine Learning
Akshay Aradhya

December 3, 2017

1 Problem 1

i)
Now we know that x is both concave and convex. Therefore 3x would also be convex. Similarly e^{y+z} is also convex. We can convert the minimum function to a maximum by using max(x) = -min(-x). Hence $-min(-x^2, \log(y))$ becomes $max(x^2, -\log(y))$ which is concave. Therefore f(x,y,z) is concave.

ii)
We find the Hessian matrix for the given function. The required partial derivatives are:

$$\frac{\partial f^2}{\partial x^2} = 6yx - 4y \quad \frac{\partial f^2}{\partial x \partial y} = 3x^2 - 4x \quad \frac{\partial f^2}{\partial y^2} = 0 \quad \frac{\partial f^2}{\partial y \partial x} = 3x^2 - 4x \tag{1}$$

Substituting these values in the Hessian matrix H, we get:

$$\begin{bmatrix} 6yx - 4y & 3x^2 - 4x \\ 3x^2 - 4x & 0 \end{bmatrix}$$

Now for H to be positive semidefinite, $\forall$ w $\in \mathbb{R}^2$ the product $wHw^T \geq 0$. Let the vector be [1, 1]. Then wHw^T is:

$$6yx - 4y + 2(3x^2 - 4x)$$

As x,y $\in (-10, 10)$, we pick any value of x, y and show that the product is < 0.
On taking, x=8 and y=-9 we get, the product to be -76. Hence the Hessian matrix is not positive semidefinite. Therefore the f(x,y) is not convex.

iii)
We find the second order derivative of the function.

$$f(x) = \log x + x^3 \quad f'(x) = \frac{1}{x} + 3x^2 \quad f''(x) = -\frac{1}{x^2} + 6x$$

Now as the domain of the function is $(1, \infty)$ we have $-\frac{1}{x^2} < 6x$. Therefore as $f''(x) \geq 0$ the function is convex.

iv)
We again convert the minimum function to a maximum one. f(x) is then equivalent to $max(-2\log 2x, x^2 - 4x + 32)$
Now, $2x$ is convex as x is convex(also concave). Also, as logarithmic function is concave, hence $-2 * \log 2x$ is convex. Similarly, $-4 * x$ is convex and x^2 and the constant 32 are also convex. Therefore $x^2 - 4x + 32$ is convex.
Hence, f(x) is convex

2 Problem 2

Let $f_1(x)$ and $f_2(x)$ be convex functions. Then, by the definition of convexity we have,

$$\lambda f_1(x) + (1-\lambda)f_1(y) \geq f_1(\lambda x + (1-\lambda)y) \ \ \forall \ \ \lambda \in [0,1] \tag{2}$$

$$\lambda f_2(x) + (1-\lambda)f_2(y) \geq f_2(\lambda x + (1-\lambda)y) \ \ \forall \ \ \lambda \in [0,1] \tag{3}$$

Let λ in equation (2) and (3) be the same, then on adding the two equations we get,

$$\lambda(f_1(x) + f_2(x)) + (1-\lambda)(f_1(y) + f_2(y)) \geq f_1(\lambda x + (1-\lambda)y) + f_2(\lambda x + (1-\lambda)y) \ \ for \ \ \lambda \in [0,1]$$

$$\Rightarrow \lambda h(x) + (1-\lambda)h(y) \geq h(\lambda x + (1-\lambda)y) \ \ for \ \ \lambda \in [0,1]$$

As $\lambda \in$ [0,1] is arbitrary, therefore the result holds for all $\lambda \in$ [0,1]. Hence h is also a convex function. i.e. sum of two convex functions is also convex.

3 Problem 3

We will disprove the given function using an example.
Let $f_1(x) = x$ and $f_2(x) = x^2$. Then the product, $g(x) = f_1(x).f_2(x) = x^3$. Now we know that x is a function that is convex and x^2 is also a convex function. However, x^3 is not a convex function.
Therefore the product of two convex functions is not necessarily a convex function.

4 Problem 4

Let θ^* be a local minimum and β be the global minimum such that, $f(\beta) < f(\theta^*)$ and $\beta \neq \theta^*$.
We will show via contradiction that $f(\beta) = f(\theta^*)$ and $\beta = \theta^*$. Now we already know that

$$f(\beta) < f(\theta^*) \tag{4}$$

From the definition of convexity, we have

$$\lambda f(\beta) + (1-\lambda)f(\theta^*) \geq f(\lambda\beta + (1-\lambda)\theta^*) \ \ \forall \ \ \lambda \in [0,1] \tag{5}$$

As θ^* is the local minimum and the value $f(\lambda\beta + (1-\lambda)\theta^*)$ is in the neighborhood of $f(\theta^*)$, therefore $f(\lambda\beta + (1-\lambda)\theta^*) > f(\theta^*)$. Hence equation (5) becomes,

$$\lambda f(\beta) + (1-\lambda)f(\theta^*) > f(\theta^*) \quad \Rightarrow \lambda f(\beta) > \lambda f(\theta^*)$$

As $\lambda \geq 0$, therefore we have

$$f(\beta) > f(\theta^*) \tag{6}$$

From equation (4) and (6), we come to a contradiction, hence $f(\beta) = f(\theta^*)$ and $\beta = \theta^*$ i.e a convex function has no local minimum, only a global minimum.

06_hw_optimization_logistic_regression_v2

December 3, 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
            return 1.0 / (1 + np.exp(-t))
```

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
            N = X.shape[0]
            scores = 0
```

```
        score = np.dot(X,w)
        sig_score = sigmoid(score)
        scores = -np.dot(y,np.log(sig_score)) - np.dot((1-y),np.log(1-sig_score))
        scores = np.sum(scores)
        return scores
```

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
        X_batch = X[mini_batch_indices]
        y_batch = y[mini_batch_indices]
        N = X_batch.shape[0]
        score = np.dot(X_batch,w)
        sig_score = sigmoid(score)
        sub = sig_score - y_batch
        dw = np.dot(X_batch.T,sub)
        dw /= N
        dw += lmbda*np.linalg.norm(w[1:])
        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
    z = np.dot(X, w)
    pred = sigmoid(z)
    return pred.round()
```

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 [19]: 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 [16]: 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()
```