Machine Learning

Homework 6 solution - Shyam Arumugaswamy

2.

Let f1 be convex function

f1(ג x1 + (1 - ג) x2) <= ג f1(x1) + (1 - ג) f1(x2)

Let f2 be convex function

f2(ג x1 + (1 - ג) x2) <= ג f2(x1) + (1 - ג) f2(x2)

Let h(x) = f1(x) + f2(x)

Add the above eqns

f1(ג x1 + (1 - ג) x2) + f2(ג x1 + (1 - ג) x2) <= ג f1(x1) + (1 - ג) f1(x2) + ג f2(x1) + (1 - ג) f2(x2)

< = ג(f1(x1) + f2(x1)) + (1 - ג) (f1(x2) + f2(x2))

h(ג x1 + (1 - ג) x2) <= ג h(x1) + (1 - ג) h(x2)

hence proved

3.

If f1 and f2 are two convex functions, prove f1.f2 is convex.

We will use proof of contradiction

Let f1 be convex function such as f1(x) = 1 + x

and f2 be convex function such as f2(x) = 1 – x

so f1(x) . f2(x) = (1 + x)(1-x) =1- x^2 = h(x) …say

h(x) = 1 – x^2 is not convex function as $h''(x) < 0$

Hence by proof of contradiction, we proved that product of two convex functions is not convex.

4.

Let f : R^N→ R be convex.

If θ^* is a local minimum of f over a convex set N, then we will prove θ^* is also a global minimum of f over a convex set N.

Proof: Since N is a convex set, for any θ, $\theta - \theta^*$ is a feasible direction. Since θ^* is a local minimum, for any $\theta \in N$, we can choose a small enough $\alpha > 0$, such that

$f(\theta^*) \le f(\theta^* + \alpha(\theta - \theta^*))$ → 1

Furthermore, since f is convex, we have

$f(\theta^* + \alpha(\theta - \theta^*)) = f(\alpha\theta + (1 - \alpha)\,\theta^*) \le \alpha f(\theta) + (1 - \alpha)f(\theta^*)$ → 2

Combining (1) and (2), we have

$f(\theta^*) \le \alpha f(\theta) + (1 - \alpha)f(\theta^*)$, which implies that $f(\theta^*) \le f(\theta)$. Since θ is an arbitrary point in N, this immediately proves that θ^* is a global minimum

To prove: if $\nabla f(\theta^*) = 0$ then θ^* is a global minimum.

Proof:

By definition, we have that:

$f(\theta) \ge f(\theta^*) + \nabla f(\theta^*) >(\theta - \theta^*)$ for any $\theta \in N$.

Thus, if $\nabla f(\theta^*\ ) >(\theta - \theta^*) \ge 0$, then $f(\theta) - f(\theta^*) \ge \nabla f(\theta^*) >(\theta - \theta^*) \ge 0$, which means θ^* is a global minimum of f over N

1.

iii)

f(x) = log(x) + x3

f'(x) = 1/x + 3x2

f''(x) = -1/x2 + 6x

for x in range (1,∞)

f''(x) > 0

so, f(x) is convex in (1,∞)

ii)

f(x,y) = y.x3 – 2.y.x2 + y + 4

first order partial derivatives

$\frac{\partial \mathrm{f(x,y)}}{\partial x}$ = 3x2y -4xy

$\frac{\partial \mathrm{f(x,y)}}{\partial \mathrm{y}}$ = x3 -2x2 +1

second order partial derivatives

$\frac{\partial 2\mathrm{f(x,y)}}{\partial x2}$ = 6xy -4y =

$\frac{\partial 2\mathrm{f(x,y)}}{\partial \mathrm{y}2}$ = 0

$\frac{\partial 2\mathrm{f(x,y)}}{\partial \mathrm{x}\partial \mathrm{y}}$ = 6x

Hessian matrix of f(x,y)

∇2 f(x,y) = [$\begin{matrix} 6\mathrm{xy} - 4\mathrm{y} & 6\mathrm{x} \\ 6\mathrm{x} & 0 \end{matrix}$]

For x(-10,10) and y(-10,10)

∇2 f(x,y) < 0

therefore f(x,y) is not convex in the range given

Programming assignment 6: Optimization: Logistic regression

In [16]:

```
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
```

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)

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 [17]:

```
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)
```

Task 1: Implement the sigmoid function

In [18]:

```
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
    t_sigmoid : array, arbitrary shape.
        Data after applying the sigmoid function.
    """
    # TODO
    return 1 / (1 + np.exp(-t))
```

Task 2: Implement the negative log likelihood

As defined in Eq. 33

In [19]:

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

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

In [20]:

```
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
```

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 [21]:

```
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
    X_batch = X[mini_batch_indices]
    y_batch = y[mini_batch_indices]
    N = X_batch.shape[0]
    scores = np.dot(X_batch,w)
    sigmoid_score = sigmoid(scores)
    subtract = sigmoid_score - y_batch
    dw = np.dot(X_batch.T,subtract)
    dw /= N
    dw += lmbda*np.linalg.norm(w[1:])
    return dw
```

Train the logistic regression model (nothing to do here)

In [22]:

```
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
    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
```

Task 4: Implement the function to obtain the predictions

In [23]:

```
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
    a = np.dot(X, w)
    y_pred = sigmoid(a)
    return y_pred.round()
```

Full batch gradient descent

In [24]:

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

In [25]:

```
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 [26]:

```
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 [27]:

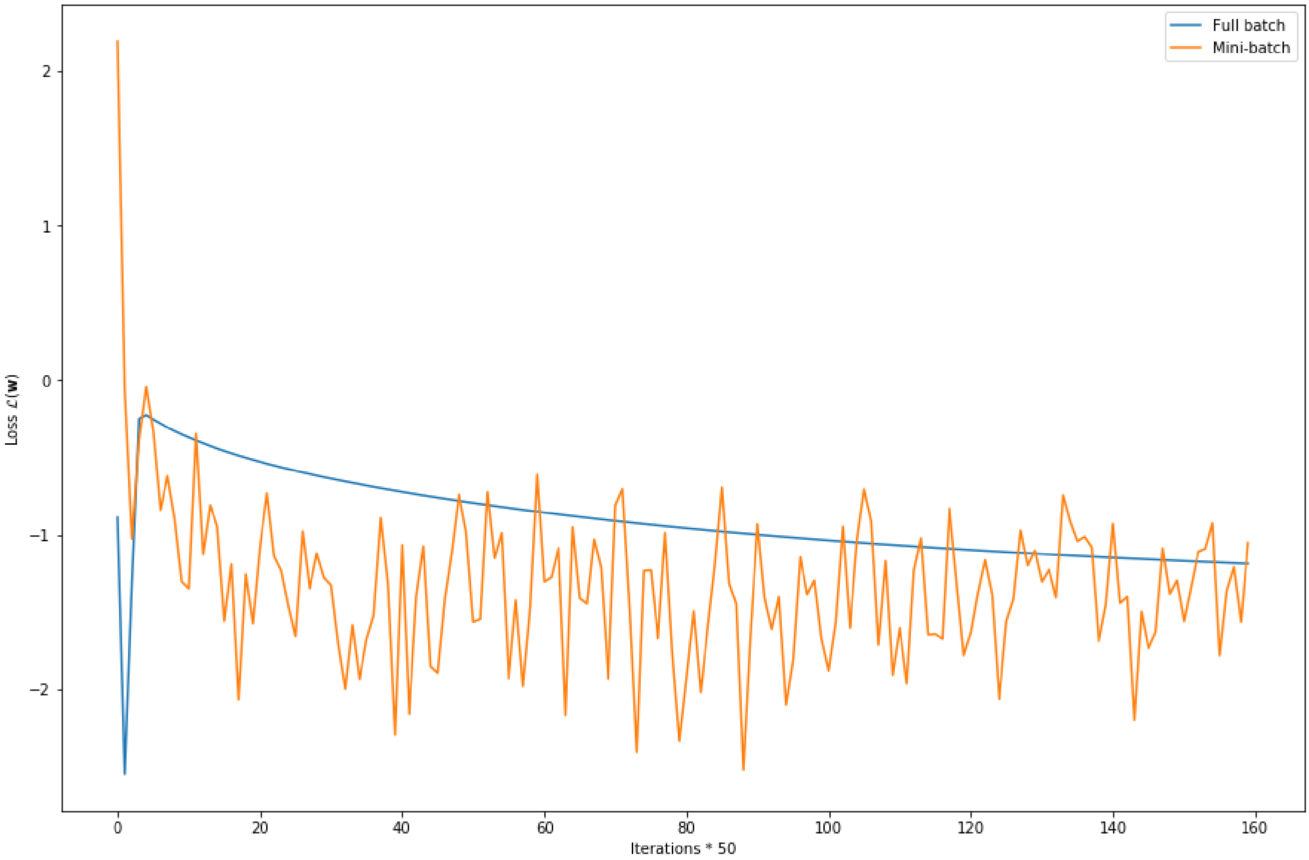
```
y_pred_full = predict(X_test, w_full)
y_pred_minibatch = predict(X_test, w_minibatch)

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
```

In [28]:

```
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
Loss $\mathcal{L}(\mathbf{w})$
Iterations * 50