Machine Learning Homework 06

Author: Jesus Arturo Sol Navarro

1 Problem 4

1.1 a)

Computing the second derivative.

$$h''(x) = g''(f(x)) \cdot (f'(x))^2 + g'(f(x)) \cdot f''(x)$$

- It can be concluded that $g''(f(x)) \geq 0$ and $f''((x) \geq 0$ since both functions are convex.
- $(f'(x))^2 \geq 0$ because squared terms are never negative.
- The term $g'(f(x))$ can either be negative, zero or positive. Since it cannot be proved that $h''(x) \geq 0$, the statement is not true in general.

1.2 b)

- Since $g(x)$ is a non-decreasing function, it can be affirmed that the slope of the tangent line at any point will be non-negative.The derivative is always $g'(x) \geq 0$
- Since there's only mulitplication and sum of non-negative terms, we can affirm that : $h''(x) \geq 0$. This statement is true : $h(x)$ is a convex function.

2 Problem 5

2.1 a)

Set Partial Derivatives to Zero

$$\frac{\partial f}{\partial x_1} = x_1 + 2 = 0 \implies x_1 = -2.$$

$$\frac{\partial f}{\partial x_2} = 2x_2 + 1 = 0 \implies x_2 = -\frac{1}{2}.$$

The critical point is:

$$x^* = (-2, -\frac{1}{2}).$$

2.2 b)

Step 1

$$\nabla f(0,0) = \begin{bmatrix} 0+2 \\ 2(0)+1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

$$x^{(1)} = x^{(0)} - \tau \nabla f(x^{(0)}) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - 1 \cdot \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix}.$$

Step 2

$$\nabla f(-2,-1) = \begin{bmatrix} -2+2 \\ 2(-1)+1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}.$$

$$x^{(2)} = x^{(1)} - \tau \nabla f(x^{(1)}) = \begin{bmatrix} -2 \\ -1 \end{bmatrix} - 1 \cdot \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} -2 \\ 0 \end{bmatrix}.$$

2.3 c)

The algorithm will never converge to $x^* = (-2, -\frac{1}{2})$. This is because, with a learning rate of $\tau = 1$,the updates oscillate indefinitely , leading to a repeating cycle.
This behavior is due to overshooting. To avoid this issue, smaller steps in the negative gradient direction are required. In other words, the learning rate needs to be smaller.

3 Problem 6

3.1 Load and preprocess the data

```
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)
```

3.2 Implement the sigmoid function

```
    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
    a= np.exp(-t)
    t_sigmoid = 1 / (1+a)
    return t_sigmoid
```

3.3 Implement the negative log likelihood

```
    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]
    D= X.shape[1]

    nll=0
    for i in range(N):  # 0 to 4
        y_i = y[i]
        a= np.dot(w.T , X[i])
        sig = sigmoid(a)
        if y_i == 1:
            log= np.log(sig)
        if y_i == 0:
```

```
            log=np.log(1-sig)

        b= -1*log
        nll= nll+ b

    return nll
```

3.4 Computing the loss function $\mathcal{L}(\mathbf{w})$

```
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
```

3.5 Implement the gradient $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$

```
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]

a = np.dot(X_batch, w)
y_hat = sigmoid(a)

error = y_hat - y_batch
dw = (np.dot(X_batch.T, error) / len(y_batch)) + (lmbda * w )

return dw
```

3.6 Train the logistic regression model

```
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
```

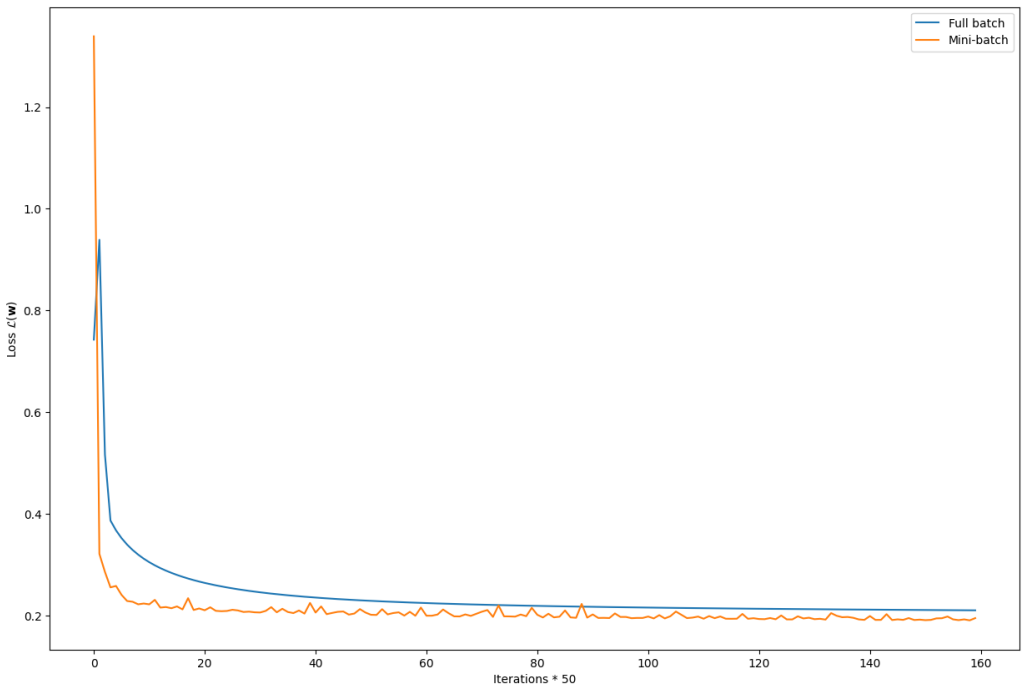
3.7 Implement the function to obtain the predictions

```
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)

    sig = sigmoid(a)

    y_pred = (sig >= 0.5).astype(int)
    return y_pred
```

3.8 Full batch gradient descent

```
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)

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)))
```



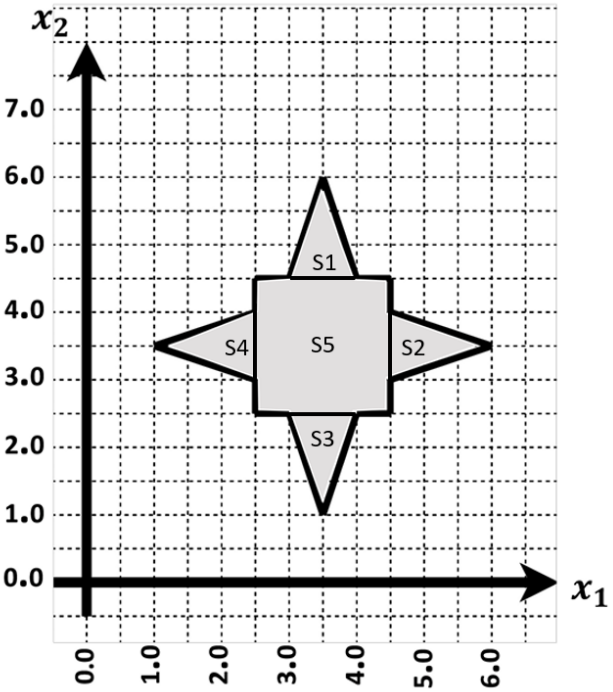
4 Problem 7

4.1 a)

The region S is not convex, because, for instance, the point $(2.25, 4.75)$ of the line segment between the points $(1, 3.5)$ and $(3.5, 6)$ does not lie inside S, which violates the condition of convexity.

4.2 b)

Splitting the region S, into convex subregions $(S_1, S_2, S_3, S_4, S_5)$. Any line segment between any two points inside or on the edge of a triangle or square lies entirely within.



Following python function can be used to determine the global minimum of $f(x_1, x_2)$ over the shaded region S. :

```
def find_global_minimum (convex_regions,convex_function)

global_minimum = float('inf')

for i in range (0,5):

    minimum = ConvOpt(convex_function,convex_regions[i])

    if  minimum < global_minimum:
        global_minimum = minimum

return global_minimum
```