Machine learning Homework- Optimisation

Abinav Ravi Venkatakrishnan - 03694216 and Abhijeet Parida - 03679676

November 24, 2018

Problem 1:

1. we know that sum of convex function is a convex function. $x^2, 2y, cos(sin(\sqrt{\pi}))$ are convex functions because linear and constant are convex.

   $-min\{-x^2, log(y)\} = max\{x^2, -log(y)\}$. The max is convex if both functions are convex which is true for our case. So, it is a convex function.

2. we know that the point-wise addition of two convex function is convex; but here $log(x)$ and $-x^3$ are concave; so overall is concave.

3. $-min\{log(3x+1), -x^4 - 3x^2 + 8x - 42\} = max\{-log(3x+1), x^4 + 3x^2 - 8x + 42\}$ The max is convex if both functions are convex which is true for our case. So, it is a convex function.

4. by plotting Fig. 1, we see that the function $-x^2y$ is not convex. So, it is not a convex function.

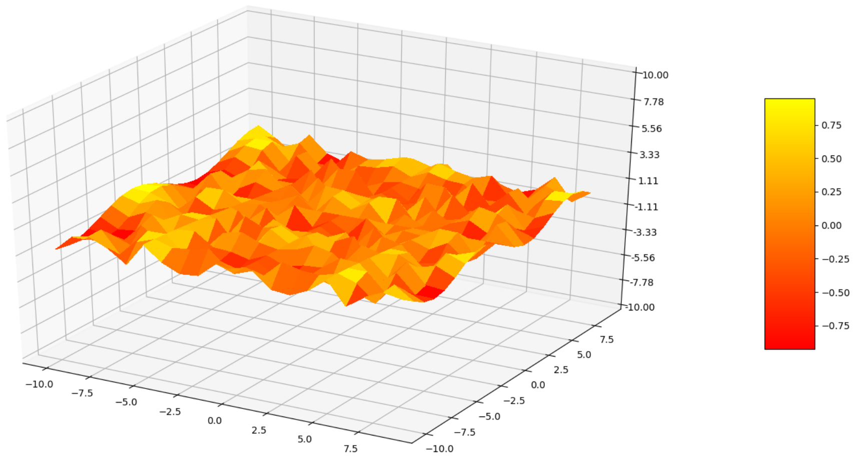

Figure 1:

Problem 2:

we know that the functions $f_1$andf_2 are convex in $\mathcal{R}$.
so,

$$\lambda_1 f_1(x_1) + (1 - \lambda_1) f_1(x_2) \geqslant f_1(\lambda_1 x_1 + (1 - \lambda_1) x_2) \tag{1}$$

and

$$\lambda_1 f_2(x_1) + (1 - \lambda_1) f_2(x_2) \geqslant f_2(\lambda_1 x_1 + (1 - \lambda_1) x_2) \tag{2}$$

$h(x) = max(f_1(x), f_2(x))$ and given $f_1 \text{and} f_2$ are convex can be written as

$$\begin{cases} f_1(x) & ; x \geqslant C \\ f_2(x) & ; x \leqslant C \end{cases}$$

for $x_1, x_2 \geqslant C$; Eqn 1 is true and for $x_1, x_2 \leqslant C$; Eqn 2 is true.
for the case $x_2 \geqslant C$ and $x_1 \leqslant C$. We have to show two scenario to be true.

$$\lambda_1 f_1(x_1) + (1 - \lambda_1) f_2(x_2) \geqslant f_1(\lambda_1 x_1 + (1 - \lambda_1) x_2) \tag{3}$$

Eq. 3 is true because Eq. 1 is true and on the LHS $(1 - \lambda_1) f_2(x_2) > (1 - \lambda_1) f_1(x_2)$ from definition of $h(x)$.

$$\lambda_1 f_1(x_1) + (1 - \lambda_1) f_2(x_2) \geqslant f_2(\lambda_1 x_1 + (1 - \lambda_1) x_2) \tag{4}$$

Eq. 4 is true because Eq. 2 is true and on the LHS $\lambda_1 f_1(x_1) > \lambda_1 f_2(x_1)$ from definition of $h(x)$.

Problem 3:

we know that $f_1 \text{and} f_2$ are convex and their exist a minima $x_1 \text{and} x_2$ such that

$$f_1'(x_1) = 0 \; ; \; f_1''(x_1) > 0 \tag{5}$$

$$f_2'(x_2) = 0 \; ; \; f_2''(x_2) > 0 \tag{6}$$

for our function $f_1(f_2(x))$, the local minima can be found at

$$f_1'(f_2(x)).f_2'(x) = 0 \tag{7}$$

From Eq. 5 and Eq. 6 the possible candidates for minima of $f_1 f_2(x)$ is $x = x_2$ and $f_2(x) = x_1$.
For the critical point $x = x_2$ is minima if $(f_1 f_2(x))'' > 0$

$$f_2''(x).f_1'(f_2(x)) + f_2'(x).f_1''(f_2(x)).f_2'(x) \tag{8}$$

we know that $f_2'(x_2).f_1''(f_2(x_2)).f_2'(x_2) = 0$ from Eq. 6 and in the term $f_2''(x).f_1'(f_2(x))$,$f_2''(x)$ is +ve from Eq. 6. Therefore, the final sign of the Eq.8 depends on the sign of $f_1'(f_2(x))$,$f_2''(x)$. Since it is positive/negative we cannot say it is convex always.

For the critical point $f_2(x) = x_1$ is minima if $(f_1 f_2(x))'' > 0$.
we know that $f_2''(x).f_1'(f_2(x)) = 0$ from Eq. 7 and in the term $f_2'(x).f_1''(f_2(x)).f_2'(x)$,$f_2''(x)$ is always +ve from Eq. 6. Therefore, the statement is true in this case.
Overall, the statement is true for some f_1, f_2 and not true for other f_2, f_1

Problem 4:

Theorem: Suppose $f : x \mapsto \mathcal{R}$ is a differentiable function and $\mathcal{R}$ is convex. Then f is convex iff for $x, y \in \mathcal{X}$ then

$$f(y) \geqslant f(x) + (y - x)^T \nabla f(x) \tag{9}$$

for $\nabla f(x) = 0$, we get $f(y) \geqslant f(x)$ hence $f(x)$ is the lowest possible value when$\nabla f(x) = 0$

Programming assignment 5: 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
```

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)

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. Download the notebook in HTML (click File > Download as > .html)
3. Convert the HTML to PDF using e.g. https://www.sejda.com/html-to-pdf or `wkhtmltopdf` for Linux (tutorial)
4. Concatenate your solutions for other tasks with the output of Step 3. 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.

This way is preferred to using `nbconvert`, since `nbconvert` clips lines that exceed page width and makes your code harder to grade.

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)
```

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

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.
    """
    # TODO
    value = sigmoid(np.dot(X,w))
    nll = np.sum((y*np.log(value))+(1-y)*(np.log(1-value)))

    return (nll)
```

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:])**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 [6]:
```
def get_gradient(X, y, w, mini_batch_indices, lmbda):
    """
    Calculates the gradient (full or mini-batch) of the negative log likelilhoo
d 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) calcu
lation 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
    n_batch = mini_batch_indices.shape[0]
    nll_gradient = np.dot(X[mini_batch_indices].T,sigmoid(np.dot(X[mini_batch_i
ndices], w)) - y[mini_batch_indices])
    ones = np.ones(w.shape)
```

```
    ones[0] = 0
    reg_gradient = lmbda * ones * w
    grad = nll_gradient / n_batch + reg_gradient
    return grad
```

Train the logistic regression model (nothing to do here)

In [7]:
```
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 r
uns 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 [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.
    """
    # TODO
    return (sigmoid(np.dot(X, w)) > 0.5).astype(np.int)
```

Full batch gradient descent

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

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)
```

```
Step 0, loss = -0.7427
Step 50, loss = -0.9390
Step 100, loss = -0.5167
Step 150, loss = -0.3868
Step 200, loss = -0.3675
Step 250, loss = -0.3522
Step 300, loss = -0.3395
Step 350, loss = -0.3288
Step 400, loss = -0.3197
Step 450, loss = -0.3117
Step 500, loss = -0.3048
Step 550, loss = -0.2986
Step 600, loss = -0.2931
Step 650, loss = -0.2882
Step 700, loss = -0.2837
Step 750, loss = -0.2796
Step 800, loss = -0.2759
Step 850, loss = -0.2725
Step 900, loss = -0.2694

Step 950, loss = -0.2665
Step 1000, loss = -0.2639
Step 1050, loss = -0.2614
Step 1100, loss = -0.2591
```

```
Step 1150, loss = -0.2569
Step 1200, loss = -0.2549
Step 1250, loss = -0.2530
Step 1300, loss = -0.2513
Step 1350, loss = -0.2496
Step 1400, loss = -0.2481
Step 1450, loss = -0.2466
Step 1500, loss = -0.2452
Step 1550, loss = -0.2439
Step 1600, loss = -0.2427
Step 1650, loss = -0.2415
Step 1700, loss = -0.2404
Step 1750, loss = -0.2393
Step 1800, loss = -0.2383
Step 1850, loss = -0.2373
Step 1900, loss = -0.2364
Step 1950, loss = -0.2356
Step 2000, loss = -0.2347
Step 2050, loss = -0.2339
Step 2100, loss = -0.2332
Step 2150, loss = -0.2325
Step 2200, loss = -0.2318
Step 2250, loss = -0.2311
Step 2300, loss = -0.2305
Step 2350, loss = -0.2299
Step 2400, loss = -0.2293
Step 2450, loss = -0.2287
Step 2500, loss = -0.2282
Step 2550, loss = -0.2276
Step 2600, loss = -0.2271
Step 2650, loss = -0.2266
Step 2700, loss = -0.2262
Step 2750, loss = -0.2257
Step 2800, loss = -0.2253
Step 2850, loss = -0.2249
Step 2900, loss = -0.2245
Step 2950, loss = -0.2241
Step 3000, loss = -0.2237
Step 3050, loss = -0.2233
Step 3100, loss = -0.2230
Step 3150, loss = -0.2226
Step 3200, loss = -0.2223
Step 3250, loss = -0.2219
Step 3300, loss = -0.2216
Step 3350, loss = -0.2213
Step 3400, loss = -0.2210
Step 3450, loss = -0.2207
Step 3500, loss = -0.2205
Step 3550, loss = -0.2202
Step 3600, loss = -0.2199
Step 3650, loss = -0.2196
Step 3700, loss = -0.2194
Step 3750, loss = -0.2191
Step 3800, loss = -0.2189
Step 3850, loss = -0.2187
Step 3900, loss = -0.2184
Step 3950, loss = -0.2182
Step 4000, loss = -0.2180
Step 4050, loss = -0.2178
Step 4100, loss = -0.2176
Step 4150, loss = -0.2174
Step 4200, loss = -0.2172
Step 4250, loss = -0.2170
Step 4300, loss = -0.2168
Step 4350, loss = -0.2166
Step 4400, loss = -0.2164
```

```
Step 4450, loss = -0.2163
Step 4500, loss = -0.2161
Step 4550, loss = -0.2159
Step 4600, loss = -0.2157
Step 4650, loss = -0.2156
Step 4700, loss = -0.2154
Step 4750, loss = -0.2153
Step 4800, loss = -0.2151
Step 4850, loss = -0.2150
Step 4900, loss = -0.2148
Step 4950, loss = -0.2147
Step 5000, loss = -0.2145
Step 5050, loss = -0.2144
Step 5100, loss = -0.2142
Step 5150, loss = -0.2141
Step 5200, loss = -0.2140
Step 5250, loss = -0.2138
Step 5300, loss = -0.2137
Step 5350, loss = -0.2136
Step 5400, loss = -0.2135
Step 5450, loss = -0.2133
Step 5500, loss = -0.2132
Step 5550, loss = -0.2131
Step 5600, loss = -0.2130
Step 5650, loss = -0.2129
Step 5700, loss = -0.2128
Step 5750, loss = -0.2126
Step 5800, loss = -0.2125
Step 5850, loss = -0.2124
Step 5900, loss = -0.2123
Step 5950, loss = -0.2122
Step 6000, loss = -0.2121
Step 6050, loss = -0.2120
Step 6100, loss = -0.2119
Step 6150, loss = -0.2118
Step 6200, loss = -0.2117
Step 6250, loss = -0.2116
Step 6300, loss = -0.2115
Step 6350, loss = -0.2114
Step 6400, loss = -0.2113
Step 6450, loss = -0.2112
Step 6500, loss = -0.2111
Step 6550, loss = -0.2110
Step 6600, loss = -0.2110
Step 6650, loss = -0.2109
Step 6700, loss = -0.2108
Step 6750, loss = -0.2107
Step 6800, loss = -0.2106
Step 6850, loss = -0.2105
Step 6900, loss = -0.2104
Step 6950, loss = -0.2104
Step 7000, loss = -0.2103
Step 7050, loss = -0.2102
Step 7100, loss = -0.2101
Step 7150, loss = -0.2100
Step 7200, loss = -0.2100
Step 7250, loss = -0.2099
Step 7300, loss = -0.2098
Step 7350, loss = -0.2097
Step 7400, loss = -0.2097
Step 7450, loss = -0.2096
Step 7500, loss = -0.2095
Step 7550, loss = -0.2094
Step 7600, loss = -0.2094
Step 7650, loss = -0.2093
Step 7700, loss = -0.2092
```

```
Step 7750, loss = -0.2091
Step 7800, loss = -0.2091
Step 7850, loss = -0.2090
Step 7900, loss = -0.2089
Step 7950, loss = -0.2089
```

In [11]:
```python
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)
```

```
Step 0, loss = -1.3392
Step 50, loss = -0.3212
Step 100, loss = -0.2856
Step 150, loss = -0.2550
Step 200, loss = -0.2577
Step 250, loss = -0.2400
Step 300, loss = -0.2279
Step 350, loss = -0.2263
Step 400, loss = -0.2212
Step 450, loss = -0.2227
Step 500, loss = -0.2210
Step 550, loss = -0.2298
Step 600, loss = -0.2145
Step 650, loss = -0.2155
Step 700, loss = -0.2132
Step 750, loss = -0.2167
Step 800, loss = -0.2108
Step 850, loss = -0.2328
Step 900, loss = -0.2096
Step 950, loss = -0.2125
Step 1000, loss = -0.2090
Step 1050, loss = -0.2148
Step 1100, loss = -0.2078
Step 1150, loss = -0.2070
Step 1200, loss = -0.2074
Step 1250, loss = -0.2097
Step 1300, loss = -0.2084
Step 1350, loss = -0.2054
Step 1400, loss = -0.2060
Step 1450, loss = -0.2047
Step 1500, loss = -0.2043
Step 1550, loss = -0.2074
Step 1600, loss = -0.2148
Step 1650, loss = -0.2046
Step 1700, loss = -0.2115
Step 1750, loss = -0.2051
Step 1800, loss = -0.2030
Step 1850, loss = -0.2080
Step 1900, loss = -0.2019
Step 1950, loss = -0.2228
Step 2000, loss = -0.2040
Step 2050, loss = -0.2160
Step 2100, loss = -0.2008

Step 2150, loss = -0.2031
Step 2200, loss = -0.2053
Step 2250, loss = -0.2059
Step 2300, loss = -0.1998
Step 2350, loss = -0.2019
Step 2400, loss = -0.2105
Step 2450, loss = -0.2038
```

```
Step 2500, loss = -0.1994
Step 2550, loss = -0.1990
Step 2600, loss = -0.2104
Step 2650, loss = -0.2001
Step 2700, loss = -0.2027
Step 2750, loss = -0.2041
Step 2800, loss = -0.1976
Step 2850, loss = -0.2050
Step 2900, loss = -0.1973
Step 2950, loss = -0.2133
Step 3000, loss = -0.1973
Step 3050, loss = -0.1973
Step 3100, loss = -0.1996
Step 3150, loss = -0.2095
Step 3200, loss = -0.2022
Step 3250, loss = -0.1960
Step 3300, loss = -0.1958
Step 3350, loss = -0.1998
Step 3400, loss = -0.1969
Step 3450, loss = -0.2010
Step 3500, loss = -0.2050
Step 3550, loss = -0.2082
Step 3600, loss = -0.1948
Step 3650, loss = -0.2173
Step 3700, loss = -0.1959
Step 3750, loss = -0.1957
Step 3800, loss = -0.1953
Step 3850, loss = -0.1993
Step 3900, loss = -0.1961
Step 3950, loss = -0.2125
Step 4000, loss = -0.1988
Step 4050, loss = -0.1935
Step 4100, loss = -0.2007
Step 4150, loss = -0.1936
Step 4200, loss = -0.1950
Step 4250, loss = -0.2072
Step 4300, loss = -0.1935
Step 4350, loss = -0.1927
Step 4400, loss = -0.2200
Step 4450, loss = -0.1930
Step 4500, loss = -0.1993
Step 4550, loss = -0.1923
Step 4600, loss = -0.1925
Step 4650, loss = -0.1921
Step 4700, loss = -0.2012
Step 4750, loss = -0.1943
Step 4800, loss = -0.1942
Step 4850, loss = -0.1917
Step 4900, loss = -0.1922
Step 4950, loss = -0.1922
Step 5000, loss = -0.1950
Step 5050, loss = -0.1912
Step 5100, loss = -0.1977
Step 5150, loss = -0.1912
Step 5200, loss = -0.1955
Step 5250, loss = -0.2047
Step 5300, loss = -0.1982
Step 5350, loss = -0.1918
Step 5400, loss = -0.1927
Step 5450, loss = -0.1947
Step 5500, loss = -0.1905
Step 5550, loss = -0.1957
Step 5600, loss = -0.1915
Step 5650, loss = -0.1949
Step 5700, loss = -0.1904
Step 5750, loss = -0.1903
```

```
Step 5800, loss = -0.1904
Step 5850, loss = -0.1998
Step 5900, loss = -0.1902
Step 5950, loss = -0.1913
Step 6000, loss = -0.1897
Step 6050, loss = -0.1894
Step 6100, loss = -0.1916
Step 6150, loss = -0.1892
Step 6200, loss = -0.1966
Step 6250, loss = -0.1889
Step 6300, loss = -0.1888
Step 6350, loss = -0.1949
Step 6400, loss = -0.1907
Step 6450, loss = -0.1921
Step 6500, loss = -0.1893
Step 6550, loss = -0.1900
Step 6600, loss = -0.1884
Step 6650, loss = -0.2011
Step 6700, loss = -0.1959
Step 6750, loss = -0.1930
Step 6800, loss = -0.1935
Step 6850, loss = -0.1918
Step 6900, loss = -0.1886
Step 6950, loss = -0.1877
Step 7000, loss = -0.1952
Step 7050, loss = -0.1876
Step 7100, loss = -0.1877
Step 7150, loss = -0.1989
Step 7200, loss = -0.1873
Step 7250, loss = -0.1886
Step 7300, loss = -0.1876
Step 7350, loss = -0.1911
Step 7400, loss = -0.1873
Step 7450, loss = -0.1880
Step 7500, loss = -0.1870
Step 7550, loss = -0.1875
Step 7600, loss = -0.1904
Step 7650, loss = -0.1907
Step 7700, loss = -0.1941
Step 7750, loss = -0.1884
Step 7800, loss = -0.1869
Step 7850, loss = -0.1884
Step 7900, loss = -0.1865
Step 7950, loss = -0.1908
```

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
_minibatch)))
```


```
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()
```