Problem 4:

a. False, assuming $f(x) = x^2$, $g(z) = (z-1)^2$

$h(x) = g(f(x)) = (x^2-1)^2$

$$\frac{d^2 h(x)}{dx^2} = \frac{d\, 2(x^2-1)\cdot 2x}{dx} = \frac{d(4x^3-2x)}{dx} = 12x^2-2$$

if $x=0$, $\frac{d^2 h(x)}{dx^2} = -2 < 0$

$h(x)$ is not convex

b. True. $f(x)$ is convex

$\Rightarrow \lambda f(x_0) + (1-\lambda) f(x_1) \geq f(\lambda x_0 + (1-\lambda) x_1) \quad \lambda \in [0,1]$

$g(x)$ is nondecreasing

$\Rightarrow g(\lambda f(x_0) + (1-\lambda) f(x_1)) \geq g(f(\lambda x_0 + (1-\lambda) x_1))$

$g(x)$ is convex

$\Rightarrow g(\lambda f(x_0) + (1-\lambda) f(x_1)) \leq \lambda g(f(x_0)) + (1-\lambda) g(f(x_1))$

$\Rightarrow g(f(\lambda x_0 + (1-\lambda) x_1)) \leq \lambda g(f(x_0)) + (1-\lambda) g(f(x_1))$

$h(\lambda x_0 + (1-\lambda) x_1) \leq \lambda h(x_0) + (1-\lambda) h(x_1)$

$\Rightarrow h(x)$ is convex

Problem 5:

a). $f(x_1, x_2)$ is sum of convex function, so $f(x_1, x_2)$ is convex

$$\frac{\partial f(x_1,x_2)}{\partial x_1} = x_1 + 2 = 0$$

$\Rightarrow x_1 = -2$

$$\frac{\partial f(x_1,x_2)}{\partial x_2} = 2x_2 + 1 = 0$$

$\Rightarrow x_2 = -0.5 \qquad x^* = [-2, -0.5]$

b). $X^{(0)} = (0,0) \Rightarrow x_1^{(0)} = 0,\ x_2^{(0)} = 0$

$$\frac{\partial f(x_1,x_2)}{\partial x_1}\Big|_{x_1=0} = 2$$

$$\frac{\partial f(x_1,x_2)}{\partial x_2}\Big|_{x_2=0} = 1$$

$$\begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \end{pmatrix} = \begin{pmatrix} x_1^{(0)} \\ x_2^{(0)} \end{pmatrix} - \tau \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} - 1 \cdot \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ -1 \end{pmatrix}$$

$$\frac{\partial f(x_1,x_2)}{\partial x_1}\Big|_{x_1=-2} = 0$$

$$\frac{\partial f(x_1,x_2)}{\partial x_2}\Big|_{x_2=-1} = -1$$

$$\begin{pmatrix} x_1^{(2)} \\ x_2^{(2)} \end{pmatrix} = \begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \end{pmatrix} - \tau \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} -2 \\ -1 \end{pmatrix} - 1 \cdot \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \end{pmatrix}$$

c).

$$\frac{\partial f(x_1,x_2)}{\partial x_1}\Big|_{x_1=-2} = 0$$

$$\frac{\partial f(x_1,x_2)}{\partial x_2}\Big|_{x_2=0} = 1$$

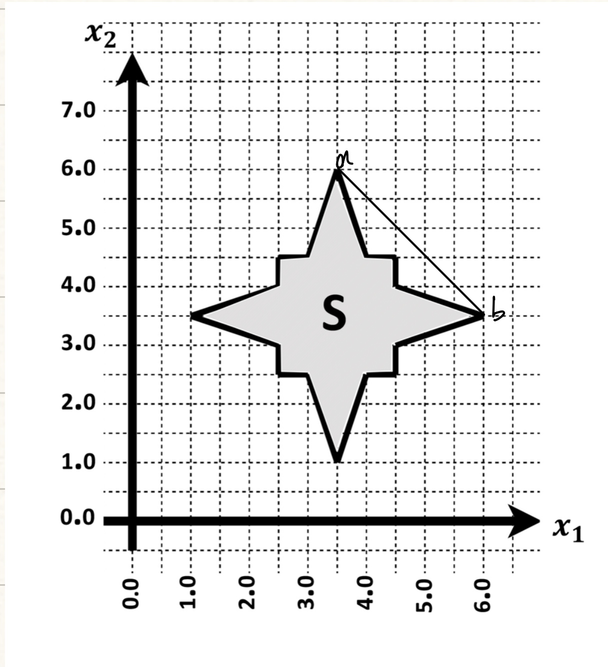
$$\begin{pmatrix} x_1^{(3)} \\ x_2^{(3)} \end{pmatrix} = \begin{pmatrix} x_1^{(2)} \\ x_2^{(2)} \end{pmatrix} - \tau \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \end{pmatrix} - 1 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ -1 \end{pmatrix} = \begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \end{pmatrix}$$

so it can't arrive the global minimum $\begin{pmatrix} -2 \\ -0.5 \end{pmatrix}$.

we can't decrease the learning rate τ to solve this problem.

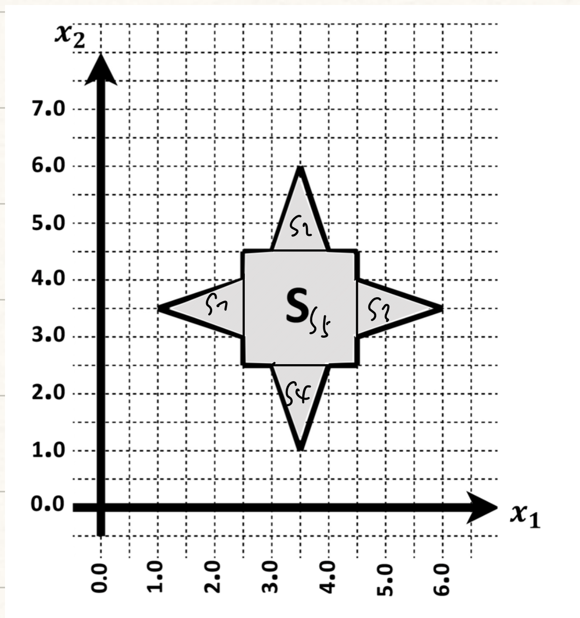
Problem 7:

a) .



there is line ab that is not in the region S. so this region is not convex

b) .



seperate the region S to $S_1 \sim S_5$

every sub region S_i is convex

global minimum of f over S is

$p^* = \min_{S_i}\{\text{CnvOpt}(f, S_i)\}$

Programming assignment 3: 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
```

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 Slide 39).

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)
```

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 / (1.0 + np.exp(-t))
```

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

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 * 0.5 * np.linalg.norm
```

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 [10]:
```
def get_gradient(X, y, w, mini_batch_indices, lmbda):
    """
    Calculates the gradient (full or mini-batch) of the negative log likelilhood 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) calculati
        This includes the full batch gradient as well, if mini_batch_indices = np.a
    lmbda: float
        Regularization strentgh. lmbda = 0 means having no regularization.

    Returns
    -------
    dw : array, shape [D]
        Gradient w.r.t. w.
    """
```

```python
    X_batch = X[mini_batch_indices]
    y_batch = y[mini_batch_indices]

    predictions = sigmoid(np.dot(X_batch, w))

    dw = X_batch.T @ (predictions - y_batch) / len(mini_batch_indices)

    dw[1:] += lmbda * w[1:]
    # TODO
    return dw
```

Train the logistic regression model (nothing to do here)

In [11]:
```python
def logistic_regression(X, y, num_steps, learning_rate, mini_batch_size, lmbda, ver
    """
    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
    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
        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 [12]:
```
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.
    """
    probabilities = sigmoid(np.dot(X, w))

    y_pred = (probabilities >= 0.5).astype(int)
    # TODO
    return y_pred
```

Full batch gradient descent

In [14]:
```
# Change this to True if you want to see loss values over iterations.
verbose = True
```

In [15]:
```
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 = 0.5168
Step 150, loss = 0.3869
Step 200, loss = 0.3676
Step 250, loss = 0.3523
Step 300, loss = 0.3396
Step 350, loss = 0.3290
Step 400, loss = 0.3198
Step 450, loss = 0.3119
Step 500, loss = 0.3050
Step 550, loss = 0.2988
Step 600, loss = 0.2934
Step 650, loss = 0.2884
Step 700, loss = 0.2840
Step 750, loss = 0.2799
Step 800, loss = 0.2763
Step 850, loss = 0.2729
Step 900, loss = 0.2698
Step 950, loss = 0.2669
Step 1000, loss = 0.2642
Step 1050, loss = 0.2618
Step 1100, loss = 0.2595
Step 1150, loss = 0.2574
Step 1200, loss = 0.2554
Step 1250, loss = 0.2535
Step 1300, loss = 0.2518
Step 1350, loss = 0.2501
Step 1400, loss = 0.2486
Step 1450, loss = 0.2471
Step 1500, loss = 0.2458
Step 1550, loss = 0.2445
Step 1600, loss = 0.2432
Step 1650, loss = 0.2421
Step 1700, loss = 0.2410
Step 1750, loss = 0.2399
Step 1800, loss = 0.2389
Step 1850, loss = 0.2380
Step 1900, loss = 0.2371
Step 1950, loss = 0.2362
Step 2000, loss = 0.2354
Step 2050, loss = 0.2346
Step 2100, loss = 0.2339
Step 2150, loss = 0.2332
Step 2200, loss = 0.2325
Step 2250, loss = 0.2318
Step 2300, loss = 0.2312
Step 2350, loss = 0.2306
Step 2400, loss = 0.2300
Step 2450, loss = 0.2295
Step 2500, loss = 0.2289
Step 2550, loss = 0.2284
Step 2600, loss = 0.2279
Step 2650, loss = 0.2275
Step 2700, loss = 0.2270
Step 2750, loss = 0.2266
```

```
Step 2800, loss = 0.2261
Step 2850, loss = 0.2257
Step 2900, loss = 0.2253
Step 2950, loss = 0.2249
Step 3000, loss = 0.2246
Step 3050, loss = 0.2242
Step 3100, loss = 0.2239
Step 3150, loss = 0.2235
Step 3200, loss = 0.2232
Step 3250, loss = 0.2229
Step 3300, loss = 0.2226
Step 3350, loss = 0.2223
Step 3400, loss = 0.2220
Step 3450, loss = 0.2217
Step 3500, loss = 0.2214
Step 3550, loss = 0.2212
Step 3600, loss = 0.2209
Step 3650, loss = 0.2206
Step 3700, loss = 0.2204
Step 3750, loss = 0.2202
Step 3800, loss = 0.2199
Step 3850, loss = 0.2197
Step 3900, loss = 0.2195
Step 3950, loss = 0.2193
Step 4000, loss = 0.2191
Step 4050, loss = 0.2189
Step 4100, loss = 0.2187
Step 4150, loss = 0.2185
Step 4200, loss = 0.2183
Step 4250, loss = 0.2181
Step 4300, loss = 0.2179
Step 4350, loss = 0.2177
Step 4400, loss = 0.2175
Step 4450, loss = 0.2174
Step 4500, loss = 0.2172
Step 4550, loss = 0.2170
Step 4600, loss = 0.2169
Step 4650, loss = 0.2167
Step 4700, loss = 0.2166
Step 4750, loss = 0.2164
Step 4800, loss = 0.2163
Step 4850, loss = 0.2161
Step 4900, loss = 0.2160
Step 4950, loss = 0.2158
Step 5000, loss = 0.2157
Step 5050, loss = 0.2156
Step 5100, loss = 0.2154
Step 5150, loss = 0.2153
Step 5200, loss = 0.2152
Step 5250, loss = 0.2151
Step 5300, loss = 0.2149
Step 5350, loss = 0.2148
Step 5400, loss = 0.2147
Step 5450, loss = 0.2146
Step 5500, loss = 0.2145
Step 5550, loss = 0.2144
```

```
Step 5600, loss = 0.2142
Step 5650, loss = 0.2141
Step 5700, loss = 0.2140
Step 5750, loss = 0.2139
Step 5800, loss = 0.2138
Step 5850, loss = 0.2137
Step 5900, loss = 0.2136
Step 5950, loss = 0.2135
Step 6000, loss = 0.2134
Step 6050, loss = 0.2133
Step 6100, loss = 0.2132
Step 6150, loss = 0.2131
Step 6200, loss = 0.2130
Step 6250, loss = 0.2129
Step 6300, loss = 0.2128
Step 6350, loss = 0.2128
Step 6400, loss = 0.2127
Step 6450, loss = 0.2126
Step 6500, loss = 0.2125
Step 6550, loss = 0.2124
Step 6600, loss = 0.2123
Step 6650, loss = 0.2122
Step 6700, loss = 0.2122
Step 6750, loss = 0.2121
Step 6800, loss = 0.2120
Step 6850, loss = 0.2119
Step 6900, loss = 0.2118
Step 6950, loss = 0.2118
Step 7000, loss = 0.2117
Step 7050, loss = 0.2116
Step 7100, loss = 0.2115
Step 7150, loss = 0.2114
Step 7200, loss = 0.2114
Step 7250, loss = 0.2113
Step 7300, loss = 0.2112
Step 7350, loss = 0.2112
Step 7400, loss = 0.2111
Step 7450, loss = 0.2110
Step 7500, loss = 0.2109
Step 7550, loss = 0.2109
Step 7600, loss = 0.2108
Step 7650, loss = 0.2107
Step 7700, loss = 0.2107
Step 7750, loss = 0.2106
Step 7800, loss = 0.2105
Step 7850, loss = 0.2105
Step 7900, loss = 0.2104
Step 7950, loss = 0.2103
```

In [16]:

```
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 = 0.3214
Step 100, loss = 0.2859
Step 150, loss = 0.2555
Step 200, loss = 0.2583
Step 250, loss = 0.2407
Step 300, loss = 0.2287
Step 350, loss = 0.2272
Step 400, loss = 0.2222
Step 450, loss = 0.2237
Step 500, loss = 0.2221
Step 550, loss = 0.2309
Step 600, loss = 0.2157
Step 650, loss = 0.2168
Step 700, loss = 0.2145
Step 750, loss = 0.2180
Step 800, loss = 0.2122
Step 850, loss = 0.2342
Step 900, loss = 0.2111
Step 950, loss = 0.2140
Step 1000, loss = 0.2105
Step 1050, loss = 0.2163
Step 1100, loss = 0.2093
Step 1150, loss = 0.2086
Step 1200, loss = 0.2091
Step 1250, loss = 0.2114
Step 1300, loss = 0.2101
Step 1350, loss = 0.2071
Step 1400, loss = 0.2078
Step 1450, loss = 0.2064
Step 1500, loss = 0.2061
Step 1550, loss = 0.2092
Step 1600, loss = 0.2166
Step 1650, loss = 0.2065
Step 1700, loss = 0.2134
Step 1750, loss = 0.2070
Step 1800, loss = 0.2049
Step 1850, loss = 0.2100
Step 1900, loss = 0.2039
Step 1950, loss = 0.2248
Step 2000, loss = 0.2060
Step 2050, loss = 0.2180
Step 2100, loss = 0.2028
Step 2150, loss = 0.2052
Step 2200, loss = 0.2074
Step 2250, loss = 0.2080
Step 2300, loss = 0.2019
Step 2350, loss = 0.2040
Step 2400, loss = 0.2127
Step 2450, loss = 0.2060
Step 2500, loss = 0.2016
Step 2550, loss = 0.2013
Step 2600, loss = 0.2126
Step 2650, loss = 0.2024
Step 2700, loss = 0.2050
Step 2750, loss = 0.2064
```

```
Step 2800, loss = 0.2000
Step 2850, loss = 0.2074
Step 2900, loss = 0.1997
Step 2950, loss = 0.2157
Step 3000, loss = 0.1997
Step 3050, loss = 0.1998
Step 3100, loss = 0.2021
Step 3150, loss = 0.2119
Step 3200, loss = 0.2046
Step 3250, loss = 0.1985
Step 3300, loss = 0.1983
Step 3350, loss = 0.2023
Step 3400, loss = 0.1994
Step 3450, loss = 0.2036
Step 3500, loss = 0.2076
Step 3550, loss = 0.2109
Step 3600, loss = 0.1975
Step 3650, loss = 0.2200
Step 3700, loss = 0.1985
Step 3750, loss = 0.1984
Step 3800, loss = 0.1980
Step 3850, loss = 0.2021
Step 3900, loss = 0.1988
Step 3950, loss = 0.2152
Step 4000, loss = 0.2016
Step 4050, loss = 0.1963
Step 4100, loss = 0.2036
Step 4150, loss = 0.1965
Step 4200, loss = 0.1978
Step 4250, loss = 0.2101
Step 4300, loss = 0.1964
Step 4350, loss = 0.1957
Step 4400, loss = 0.2229
Step 4450, loss = 0.1960
Step 4500, loss = 0.2022
Step 4550, loss = 0.1953
Step 4600, loss = 0.1955
Step 4650, loss = 0.1952
Step 4700, loss = 0.2042
Step 4750, loss = 0.1973
Step 4800, loss = 0.1973
Step 4850, loss = 0.1949
Step 4900, loss = 0.1954
Step 4950, loss = 0.1953
Step 5000, loss = 0.1981
Step 5050, loss = 0.1944
Step 5100, loss = 0.2009
Step 5150, loss = 0.1944
Step 5200, loss = 0.1987
Step 5250, loss = 0.2079
Step 5300, loss = 0.2014
Step 5350, loss = 0.1951
Step 5400, loss = 0.1960
Step 5450, loss = 0.1981
Step 5500, loss = 0.1938
Step 5550, loss = 0.1990
```

```
Step 5600, loss = 0.1949
Step 5650, loss = 0.1983
Step 5700, loss = 0.1938
Step 5750, loss = 0.1937
Step 5800, loss = 0.1939
Step 5850, loss = 0.2032
Step 5900, loss = 0.1937
Step 5950, loss = 0.1948
Step 6000, loss = 0.1932
Step 6050, loss = 0.1929
Step 6100, loss = 0.1952
Step 6150, loss = 0.1928
Step 6200, loss = 0.2002
Step 6250, loss = 0.1925
Step 6300, loss = 0.1925
Step 6350, loss = 0.1985
Step 6400, loss = 0.1944
Step 6450, loss = 0.1958
Step 6500, loss = 0.1930
Step 6550, loss = 0.1937
Step 6600, loss = 0.1921
Step 6650, loss = 0.2048
Step 6700, loss = 0.1996
Step 6750, loss = 0.1968
Step 6800, loss = 0.1973
Step 6850, loss = 0.1956
Step 6900, loss = 0.1925
Step 6950, loss = 0.1916
Step 7000, loss = 0.1991
Step 7050, loss = 0.1915
Step 7100, loss = 0.1916
Step 7150, loss = 0.2028
Step 7200, loss = 0.1913
Step 7250, loss = 0.1926
Step 7300, loss = 0.1916
Step 7350, loss = 0.1951
Step 7400, loss = 0.1913
Step 7450, loss = 0.1920
Step 7500, loss = 0.1910
Step 7550, loss = 0.1915
Step 7600, loss = 0.1945
Step 7650, loss = 0.1948
Step 7700, loss = 0.1982
Step 7750, loss = 0.1926
Step 7800, loss = 0.1911
Step 7850, loss = 0.1925
Step 7900, loss = 0.1906
Step 7950, loss = 0.1950
```

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 [17]:
```
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_min
```

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

In [18]:
```
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()
```