THEODOR STOICAN 03725732
LUKAS KOEBE 03723950
KEWIN HAWRYLUK 03692265

Assignment 6

Problem 1

a) $g_1 : \mathbb{R}^{d_1} \to \mathbb{R}$ – convex

$g_2 : \mathbb{R} \to \mathbb{R}$ – convex

$h(x) = g_2(g_1(x))$ – convex?

Product rule

$$\boxed{(fg)' = f'g + fg'}$$

$$h'(x) = g_2'(g_1(x)) \cdot g_1'(x)$$

$$h''(x) = \underbrace{g_2''(g_1(x))}_{\geq 0} \cdot \underbrace{g_1'(x)}_{\geq 0}^2 + \underbrace{g_2'(g_1(x))}_{?} \cdot \underbrace{g_1''(x)}_{\geq 0}$$

From the expression above, we notice that if g_2 is decreasing and has a sufficiently high negative slope, then it could be the case that $h''(x) < 0$ and, hence, non-convex.

b) From the same expression, we notice that if g_2 is non-decreasing, then $h''(x) \geq 0$ and hence convex.

c) One observation is that this expression makes sense iff $d_1 = d_2 = \cdots = d_n$.

We know that max is a convexity-preserving operation. So, $\max(g_1(x), g_2(x))$ is convex.

Then $\overbrace{\max(\underbrace{\max(g_1(x), g_2(x))}_{\text{convex}}, \underbrace{g_3(x)}_{\text{convex}})}^{\text{convex}} =$

$= \max(g_1(x), g_2(x), g_3(x))$ is also convex.

By mathematical induction, we can then prove that $\max(g_1, \ldots, g_n)$ is also convex.

Problem 2

a) $f(x_1, x_2) = 0.5x_1^2 + x_2^2 + 2x_1 + x_2 + \cos(\sin(\sqrt{\pi}))$

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

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

$$\Rightarrow x^* = \left[-2, -\frac{1}{2}\right] \text{ - the minimizer}$$

b) $\sigma = 1$, $x^{(0)} = (0, 0)$

First iteration:

- Compute the gradient in $(0,0)$:

$$\nabla f = \begin{bmatrix} x_1 + 2 & 2x_2 + 1 \end{bmatrix}$$

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

$$x^{(1)} = x^{(0)} - \sigma \nabla f(0,0) = (0,0) - (2,1)$$
$$= (-2, -1)$$

- Update x

Second iteration:

$$\nabla f(-2,-1) = [0,-1]$$

$$x^{(2)} = x^{(1)} - \sigma \nabla f(-2,-1) = (-2,-1) - (0,-1)$$
$$= (-2, 0)$$

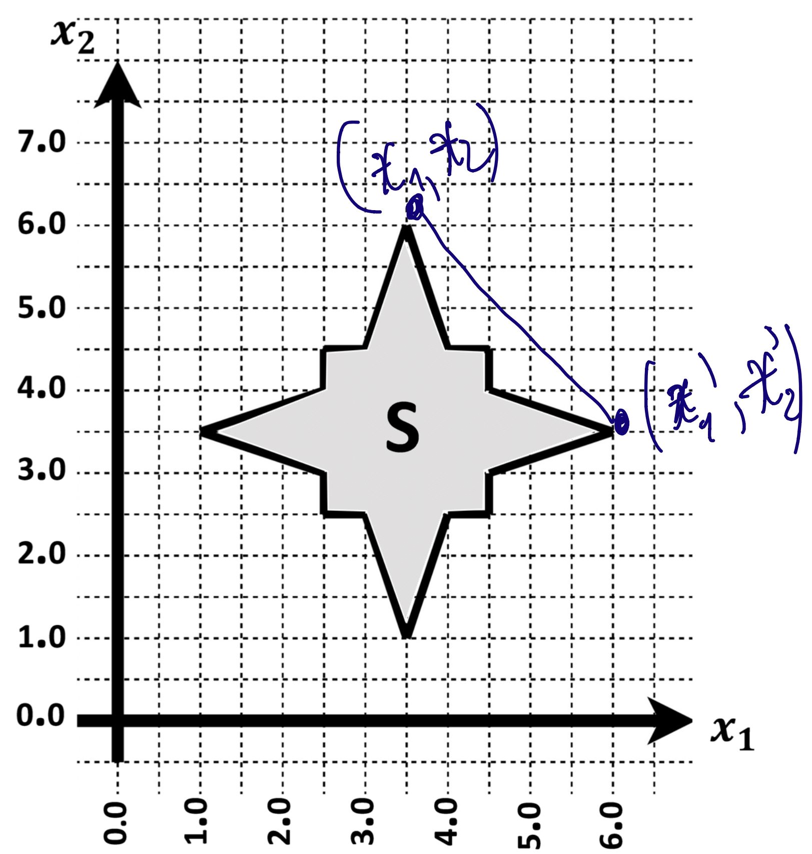
c) The gradient descent procedure from b) will never converge to the minimum, since, on the second coordinate, the gradient is too big (the function is steep) and the learning rate is also too large ($\overset{\text{"}\sigma\text{"}}{4} \cdot \overset{\text{"}\nabla f_y\text{"}}{1} = 1$, this is the least amount that will be added/subtracted from 0, being impossible to reach the min: $-\frac{1}{2}$).

One potential solution would be to make the learning rate smaller such that we could subtract/add a sufficiently small number in order to get $-\frac{1}{2}$.

Problem 4

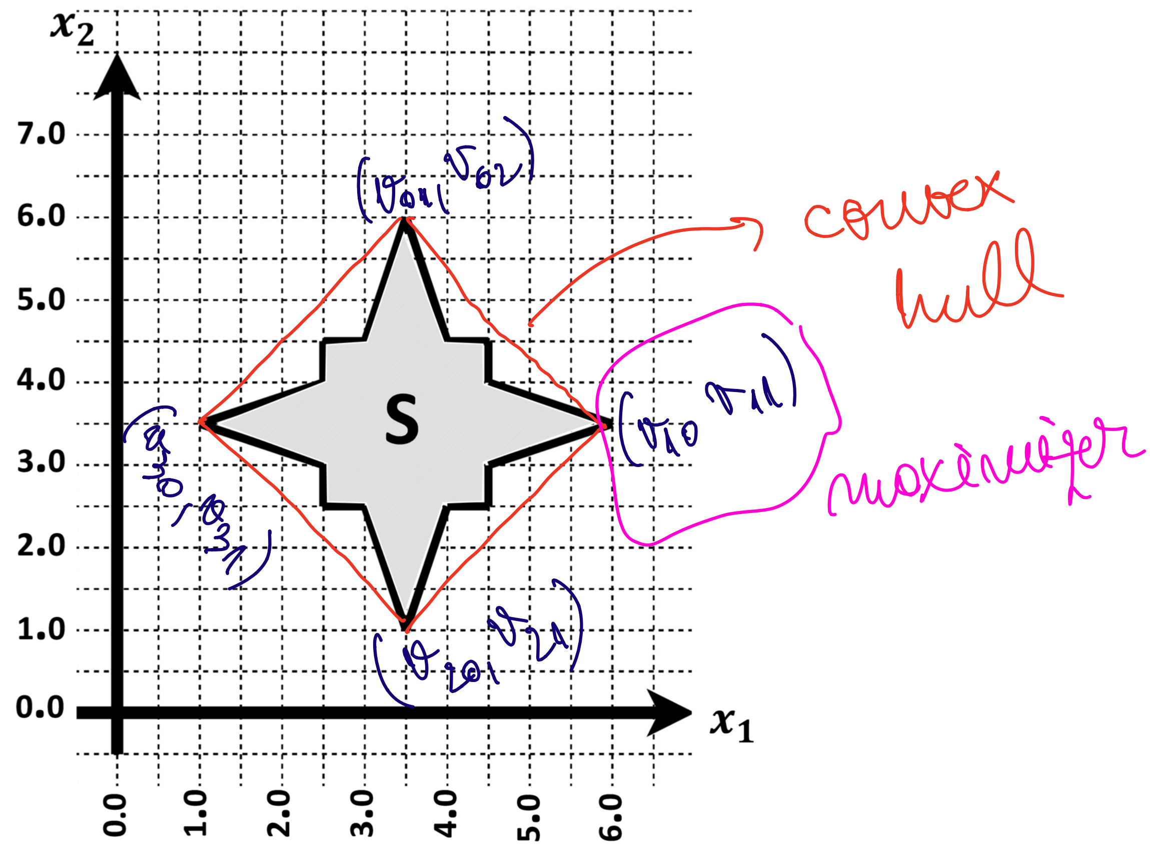
$$f(x_1, x_2) = e^{x_1 + x_2} - 5 \log x_2$$

a)



The region is not convex, since by choosing the 2 points shown above, there are points on that line that are not in the region and, hence, that violate the convexity definition ($\lambda x + (1-\lambda) y \in S$ for $x, y \in S$)

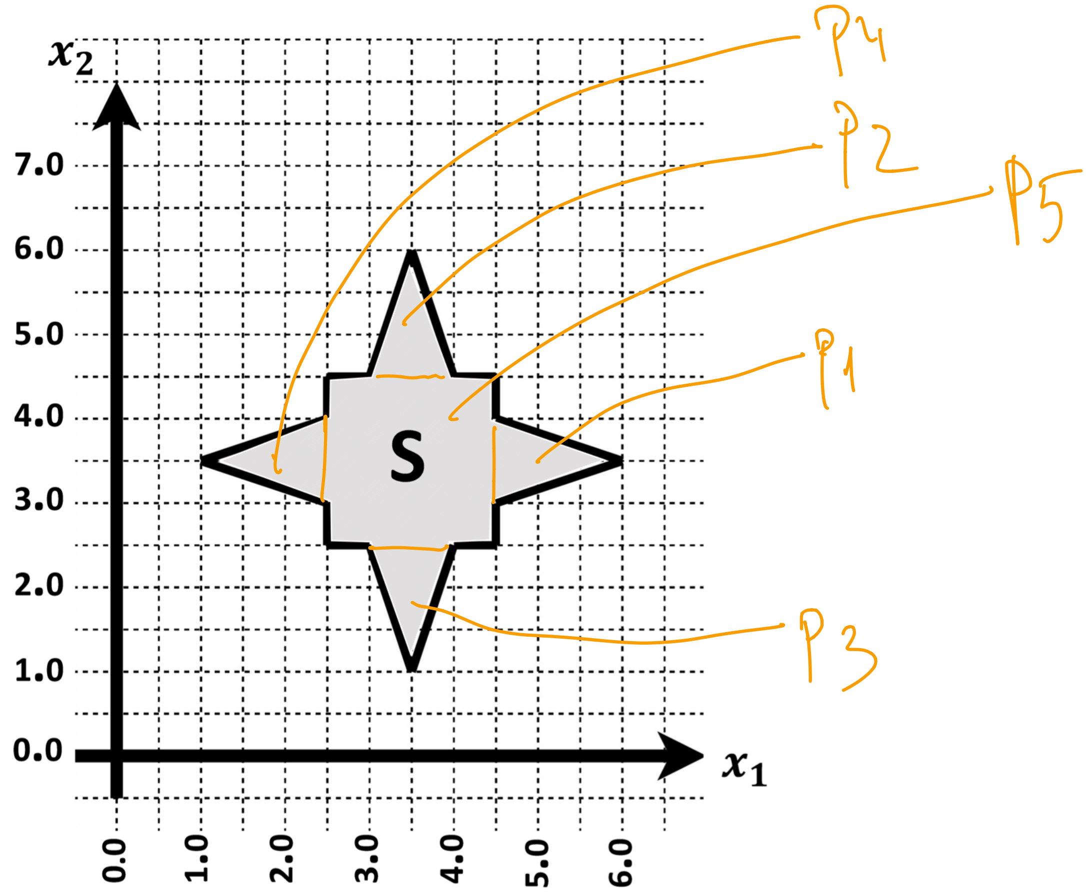
b) For getting the maximum, it is sufficient to check only the vertices of the convex hull.



There are 4 vertices on the convex hull that we need to check. However, intuitively, (v_{10}, v_{11}) is the maximum since $x_1 + x_2$ reaches its maximum in ~~this~~ point and x_2 is small enough such that $f(x_1, x_2)$ is max.

Therefore $(6.0, 3.5)$ is the maximizer.

c)



Since ConvOpt(f, D) works only on convex surfaces, one idea would be to split our surface into multiple convex sub-surfaces, compute the minimum on each one and then find the global minimum by comparing all of the previous minima.

One such potential split is shown in the figure above in which we get 5 convex polygons. Then it's easy to compute all the 5 minima and get the smallest one.

exercise_06_optimization

November 24, 2019

1 Programming assignment 3: Optimization - Logistic Regression

[131]:
```
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 (see Eq. 33).

1.2 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. Export/download the notebook as PDF (File -> Download as -> PDF via LaTeX (.pdf)). 3. Concatenate your solutions for other tasks with the output of Step 2. 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.

Make sure you are using `nbconvert` Version 5.5 or later by running `jupyter nbconvert --version`. Older versions clip lines that exceed page width, which makes your code harder to grade.

1.3 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.

[132]:
```
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)
```

1.4 Task 1: Implement the sigmoid function

[133]:
```
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 / (1 + np.exp(-t))
```

1.5 Task 2: Implement the negative log likelihood

As defined in Eq. 33

[134]:
```
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
    log_likelihood = -np.sum(y * np.log(sigmoid(np.matmul(X, w))) + (1 - y) * np.
↪log(1 - sigmoid(np.matmul(X, w))))
    return log_likelihood
```

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

[135]:
```
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
```

1.6 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!)

[136]:
```
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
    minibatch_x = X[mini_batch_indices, :]
    minibatch_y = y[mini_batch_indices]
    diff = (sigmoid(np.matmul(minibatch_x, w)) - minibatch_y)
    diff = np.reshape(diff, (diff.shape[0], 1))
    gradient_nll = np.sum(diff * minibatch_x, axis=0) / len(mini_batch_indices)
    gradient = gradient_nll + lmbda * w
    return gradient
```

1.6.1 Train the logistic regression model (nothing to do here)

[137]:
```
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
```

1.7 Task 4: Implement the function to obtain the predictions

[138]:
```
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
    y = sigmoid(np.matmul(X, w))
    y[y >= 0.5] = 1
    y[y < 0.5] = 0
    return y
```

1.7.1 Full batch gradient descent

[139]:
```
# Change this to True if you want to see loss values over iterations.
verbose = False
```

[140]:
```
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)
```

[141]:
```
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.

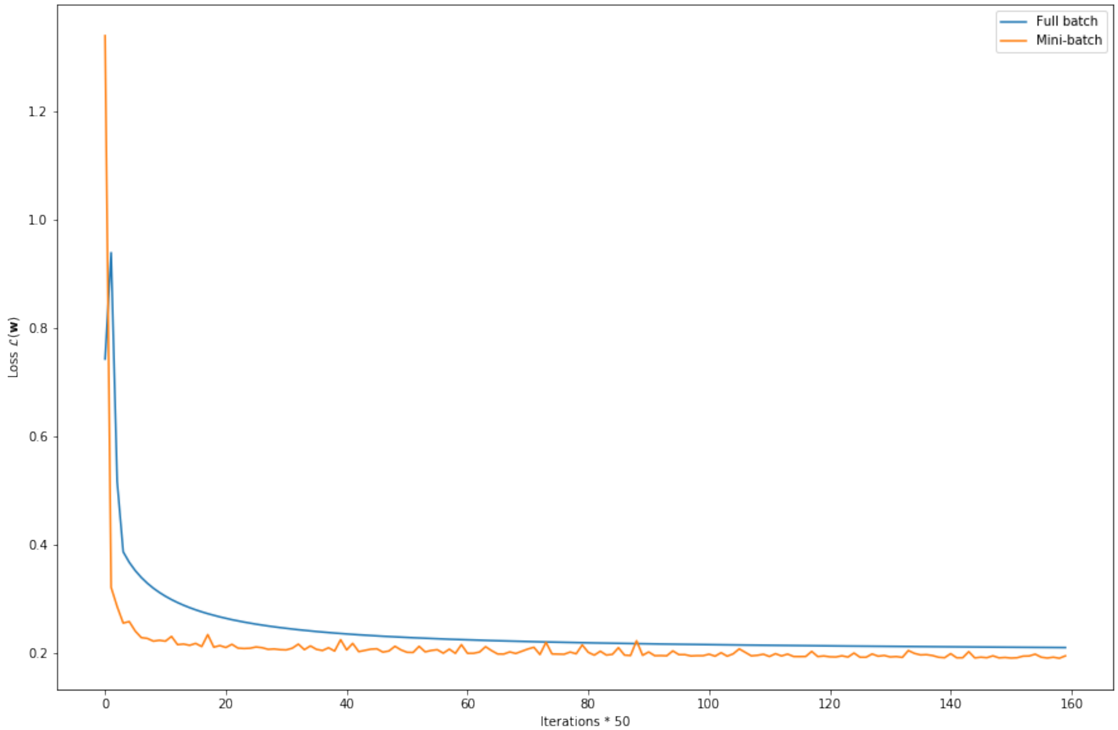
```
Full batch: accuracy: 0.9240, f1_score: 0.9384
Mini-batch: accuracy: 0.9415, f1_score: 0.9533
```

[142]:
```
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
```

[143]:
```
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
1.2
1.0
0.8
0.6
0.4
0.2
Loss $\mathcal{L}(\mathbf{w})$
0
20
40
60
80
100
120
140
160
Iterations * 50

[ ]: