Machine Learning B (2025)
Home Assignment 3

Yasin Baysal, cmv882

Contents

1 SVM with Kernels

In this exercise, we will train both a Linear SVM and a SVM with a Gaussian kernel using the dataset provided in the `non_linear_svm_data.csv` file. As discussed in the lectures, the parameter c is the misclassification penalty, and the Gaussian kernel is defined as

$$K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2a^2}\right), \quad a \in \mathbb{R}.$$

Response to Question 1. In order to report the training loss for the Linear SVM, we train a classifier using the dataset provided in the `non_linear_svm_data.csv` file given by:

x	y	label
0.0	0.0	1
0.2	0.1	1
-0.1	0.2	1
-0.2	-0.1	1
0.1	-0.2	1
0.3	0.3	0
-0.3	0.3	0
-0.3	-0.3	0
0.3	-0.3	0
0.6	0.0	0
0.0	0.6	0
-0.6	0.0	0
0.0	-0.6	0
0.5	0.5	0
-0.5	0.5	0
-0.5	-0.5	0
0.5	-0.5	0
0.1	0.0	1
0.0	0.1	1
-0.1	0.0	1

Table 1: Training data from `non_linear_svm_data.csv`.

Here, $n = 20$ is the total number of training examples and $m = 2$ is the dimensionality of each feature vector $x_i \in \mathbb{R}^2$. We encode the original labels in the table as $y_i \in \{+1, -1\}$ by mapping label 1 to $+1$ and label 0 to -1. In Figure 1, we plot the training data: "+" markers denote the $y = +1$ class and "o" markers denote the $y = -1$ class. This scatter clearly shows that the two classes are not linearly separable by any straight line.

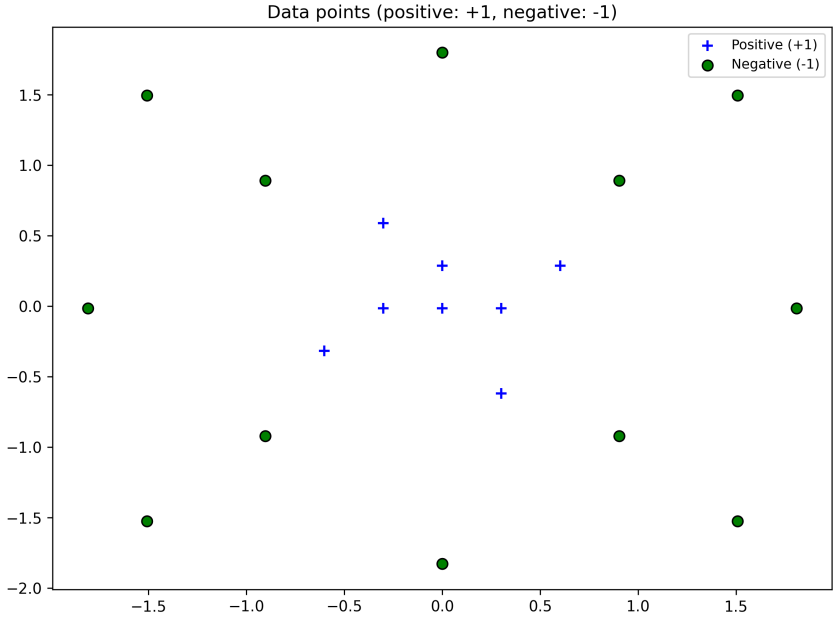

Figure 1: Scatter of the training points: "+" are the $y = +1$ class, "o" the $y = -1$ class.

Given the training dataset $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ with $x_i \in \mathbb{R}^m$ and $y_i \in \{-1, +1\}$, the soft-margin linear SVM (with $r = 1$) solves the primal optimization problem given by

$$\begin{aligned} \min_{w \in \mathbb{R}^m, b \in \mathbb{R}} \quad & \frac{1}{2}||w||^2 + c\sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & 1 - y_i(w^\top x_i + b) - \xi_i \leq 0 \\ & \xi_i \geq 0 \end{aligned}$$

for $i = 1, \ldots, n$, where $c \in \mathbb{R}_{++}$ is the misclassification penalty that we later will experiment with using the given dataset. It balances the goal of maximizing the margin with the need to reduce the classification errors. By choosing the slack variables ξ_i as

$$\xi_i = \max\{0, 1 - y_i(w^\top x_i + b)\},$$

which ensures that the both constrains are satisfied, since $\max\{0, \cdot\} \geq 0$ first ensure that the non-negativity condition $\xi_i \geq 0$ holds and by construction, $y_i(w^\top x_i + b) \geq 1 - \xi_i$ is also satisfied, then the primal optimization problem can be equivalently formulated as

$$\min_{w \in \mathbb{R}^m, b \in \mathbb{R}} \quad \frac{1}{2}\|w\|^2 + c\sum_{i=1}^{n} \max\{0, 1 - y_i(w^\top x_i + b)\}.$$

Once (w^*, b^*) is found, the training loss (primal objective value) for the linear SVM is thus

$$L_{\text{train}}(c) = \frac{1}{2}\|w^*\|^2 + c\sum_{i=1}^{n} \max\{0,\ 1 - y_i(w^{*\top} x_i + b^*)\},$$

which is a so-called hinge-loss function. We have to use a binary loss function, which has value 1 if the predicted label is incorrect and 0 if the predicted label is correct, i.e. we have

$$\frac{1}{n}\sum_{i=1}^{n} 1(y_i \neq \hat{y}_i),$$

where $\hat{y}_i = \text{Sign}(w^\top x_i + b)$ is the predicted label. Since the linear SVM QP is convex and satisfies Slater's condition, strong duality holds, so the optimal primal objective and optimal dual objective coincide. Thus, we only write the primal objective explicitly here.

To implement the linear SVM classifier in Python, we load `non_linear_svm_data.csv` into arrays X (with data from x and y) and y (labels) and map the labels by $1 \mapsto +1$ and $0 \mapsto -1$, such that $y_i \in \{-1, +1\}$. We standardize X to have zero mean and unit variance using `X = (X - X.mean(axis=0)) / X.std(axis=0)`. Then, we loop over each misclassification penalty $c \in \{1, 100, 1000\}$. For each c, we train a linear SVM using `scikit-learn`'s `LinearSVC` (Linear Support Vector Classification) algorithm by calling

```
clf = LinearSVC(
    C=c,                % misclassification penalty
    loss='hinge',       % hinge-loss SVM
)
```

Hereafter, we fit the data by calling `clf.fit(X, y)`. After fitting the data, we compute the decision scores and predicted labels `preds = np.sign(clf.decision_function(X))`. Finally, we compute the empirical binary losses as mentioned earlier from the model `loss = zero_one_loss(y, y_pred` using the `zero_one_loss()` function from `sklearn.metrics`, which calculates the proportion of misclassified points in the training dataset for us.

For different misclassification penalties $c \in \{1, 100, 1000\}$, we obtain the following losses:

Misclassification penalty c	**Training loss**
1	0.4000
100	0.4000
1000	0.4000

Table 2: Losses for Linear SVM with $r = 1$ and different misclassification penalties.

From Table 2, we see that the empirical 0–1 training loss is constant at 0.4 for all values of the misclassification penalty parameter $c \in \{1, 100, 1000\}$. Therefore, because the two classes in our dataset are not linearly separable, the number of margin-violations,

$$\sum_{i=1}^{n} \max\{0,\ 1 - y_i(w^\top x_i + b)\}$$

turns out to be a constant of 8 at the optimum hyperplane, regardless of c. This reflects the fact that our two-dimensional dataset is not linearly separable by any hyperplane, so no matter how harshly we penalize misclassification (by increasing c), the best linear decision boundary still misclassifies exactly 40% of the points, which corresponds to 8 points. In the soft-margin formulation, raising c simply scales the relative weight of the fixed set of margin-violations, but does not change which points lie on the wrong side of the hyperplane.

Response to Question 2. We now repeat the SVM-training procedure that we did in Question 1, but now we replace all the inner-products $\langle x_j, x_i \rangle$ with the Gaussian kernel (which is also known as the radial basis function (RBF)) that is given by

$$K_a(x_j, x_i) = \exp\left(-\frac{||x_j - x_i||^2}{2a^2}\right), \quad a \in \mathbb{R}.$$

For misclassification-penalty c and kernel K_a, the dual SVM problem reads

$$\begin{aligned} \max_{\lambda \in \mathbb{R}^n} \quad & \phi(\lambda) := \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{n} \lambda_i \lambda_j y_i y_j K_a(x_i, x_j) \\ \text{s.t.} \quad & 0 \leq \lambda_i \leq c \\ & \sum_{i=1}^{n} \lambda_i y_i = 0 \end{aligned}$$

for $i = 1, \ldots, n$. At optimum λ^*, the decision function (i.e., the resulting hypothesis h) can be written purely in terms of kernels, such that

$$h(x) = \text{Sign}\left(\sum_{i=1}^{n} \lambda_i^* y_i K_a(x_i, x) + b^*\right),$$

where the bias is given by $b^* = y_i - \sum_{j=1}^{n} \lambda_i^* y_j K_a(x_i, x_j)$ with $i \in [n]$, such that $0 < \lambda_i^* < c$. Once λ^* (and hence w^*, b^*) is found, the training loss for the kernel SVM is

$$L_{\text{train}}(c, a) = \sum_{i=1}^{n} \lambda_i^* - \frac{1}{2} \sum_{i,j=1}^{n} \lambda_i^* \, \lambda_j^* \, y_i \, y_j \, K_a(x_i, x_j),$$

where $K_a(x_i, x_j) = \exp\big(-\|x_i - x_j\|^2/(2a^2)\big)$. Equivalently, in its primal form,

$$L_{\text{train}}(c, a) = \frac{1}{2}\|w^*\|^2 + c \sum_{i=1}^{n} \max\left\{0, \; 1 - y_i \left(\sum_{j=1}^{n} \lambda_j^* y_j K_a(x_j, x_i) + b^*\right)\right\},$$

which is – like for the linear SVM in the previous question – a so-called hinge-loss function. However, as we are simply interested in the binary loss, we use the binary loss function from the previous question when we have to evaluate the SVM classifier.

In order to implement the SVM optimization classifier with the Gaussian kernel in Python, we reuse the arrays X and y from the last question. Then, we fix $c = 1$, and when computing the kernel, we first note that in `scikit-learn`'s `SVC` class, the RBF kernel is defined as $K(x, x') = \exp(-\gamma||x - x'||^2)$, which means that we select

$$\gamma = \frac{1}{2a^2} \Rightarrow K(x, x') = \exp(-\frac{||x - x'||^2}{2a^2}) = K_a(x, x').$$

For each kernel parameter $a \in \{0.1, 1, 10\}$, we thus compute the kernel coefficient `gamma = 1.0 / (2 * a**2)`, which we use to train a SVM with a Gaussian kernel (here called a RBF-kernel) using `scikit-learn`'s `SVC` (C-Support Vector Classification) class by calling

```
clf = SVC(
    C=c,              # misclassification penalty
    kernel='rbf',     # radial basis function kernel
    gamma=gamma,      # RBF kernel coefficient
)
```

For different $a \in \{0.1, 1, 10\}$, the corresponding values of γ are given by $\gamma \in \{50, 0.5, 0.005\}$. After having fitted the data via `clf.fit(X, y)`, we compute the predicted labels using `clf.predict(X)`, and then evaluate the 0–1 training error by again using `loss = zero_one_loss(y, clf.predict(X))` to print the fraction of misclassified training points.

For the different parameter values $a \in \{0.1, 1, 10\}$, we obtain the following training losses:

Parameter a for Gaussian kernel	**Training loss**
0.1	0.0000
1	0.0000
10	0.4000

Table 3: Training losses for SVM with Gaussian kernel for different values of parameter a.

Table 3 above demonstrates how the Gaussian kernel width parameter a affects the training loss. At $a = 0.1$ and $a = 1$ (a very narrow kernel) the SVM can perfectly separate the inner "+1" cluster from the outer "–1" ring, yielding zero binary 0–1 training loss. Thus, no points are misclassified, and the empirical misclassification count (or fraction) is zero.

As a increases to 10 the kernel becomes "flatter," and the classifier behaves more like a linear SVM – hence the training loss rises to 0.4, matching the linear-SVM error from Question 1. As $a \to \infty$, $K_a(x, x') \to 1$, and the kernel SVM degenerates to a nearly linear classifier, where the hinge-sum climbs back up to 8 (or 40%) misclassified points.

2 Logistic regression on MNIST

In this exercise, we will train a logistic regression classifier to distinguish between the handwritten digits {3, 8} using the `MNIST` dataset. The model will be trained on images of the handwritten digits labeled as 3 and 8, using ℓ_2-regularization with regularization parameter $\mu = 1$. The training will be carried out using two optimization algorithms: gradient descent with a constant learning rate $\gamma = 0.001$, and mini-batch stochastic gradient descent (SGD) with batch size $b = 10$ and the same learning rate given by $\gamma = 0.001$.

Response to Question 3. In order to load and inspect the MNIST dataset with digit images before training our classifier, we first import – along with standard libraries – `mnist` from the `keras.datasets` library. We then call the built-in data loader by `(X_train, y_train), (X_test, y_test) = mnist.load_data()` that returns two tuples:

- one for training (60,000 images and labels in `X_train` and `y_train`)
- one for testing (10,000 images and labels in `X_test` and `y_test`)

Each image is a 28×28 array of grayscale pixel values, and each label is the corresponding digit (0–9). To quickly verify that we have loaded the data correctly, we pick nine random indices from the training dataset, titling each subplot with its true class from `y_train`:

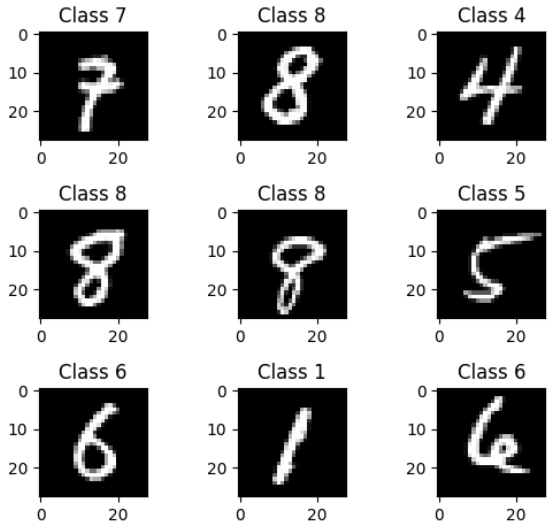

Figure 2: Random sample of nine `MNIST` training images with their true labels.

Then, each `MNIST` image is reshaped into a column–vector $x_i \in \mathbb{R}^m$, where $m = 28 \times 28 = 784$, and its labels $y_i \in \{3, 8\}$ is converted to a binary target given by

$$\tilde{y}_i = \begin{cases} 1 & \text{if } y_i = 8 \\ 0 & \text{if } y_i = 3. \end{cases}$$

We collect the n training samples as $S = \{(x_1, \tilde{y}_1), \ldots, (x_n, \tilde{y}_n)\}$, each x_i being the 784-dimensional image vector and $\tilde{y}_i \in \{0, 1\}$, so $n = |S|$, and introduce a weight vector $w \in \mathbb{R}^m$ and a bias term $b \in \mathbb{R}$ that we need to learn. For each x_i, we define the score z_i and the predicted probability p_i (i.e., the probability that the true label is 1 for x_i) by

$$z_i = w^\top x_i + b, \quad p_i = \text{Sigmoid}(z_i) = \frac{1}{1 + e^{-z_i}},$$

For each sample (x_i, y_i) for $i = 1, \ldots, n$, we define the cross-entropy loss by

$$\begin{aligned} \ell_i(w, b) &= -\left[\tilde{y}_i \log\left(\frac{1}{1 + e^{-z_i}}\right) + (1 - \tilde{y}_i) \log\left(\frac{e^{-z_i}}{1 + e^{-z_i}}\right)\right] \\ &= -\left[\tilde{y}_i \log(p_i) + (1 - \tilde{y}_i) \log(1 - p_i)\right]. \end{aligned}$$

Since we are training a logistic regression model with l_2-regularization, we add an l_2-penalty to the model with strength $\mu = 1$, giving the regularized empirical risk

$$\begin{aligned} \mathcal{L}(w, b) &= \frac{1}{n} \sum_{i=1}^{n} \ell_i(w, b) + \frac{1}{2} \left(||w||^2 + b^2\right) \\ &= \frac{1}{n} \sum_{i=1}^{n} \left[-\tilde{y}_i \log(p_i) - (1 - \tilde{y}_i) \log(1 - p_i)\right] + \frac{1}{2} \left(||w||^2 + b^2\right), \end{aligned}$$

which then gives us the regularized empirical risk minimization (ERM) problem by

$$\min_{w \in \mathbb{R}^m, b \in \mathbb{R}} \quad \mathcal{L}(w, b) = \frac{1}{n} \sum_{i=1}^{n} \left[-\tilde{y}_i \log(p_i) - (1 - \tilde{y}_i) \log(1 - p_i)\right] + \frac{1}{2} \left(||w||^2 + b^2\right),$$

From the lecture slides, we know that the loss function $\mathcal{L}(w, b)$ is both differentiable and convex. To compute the gradient of the loss function $\mathcal{L}(w, b)$, we first note that

$$\frac{\partial \ell_i}{\partial z_i} = p_i - \tilde{y}_i \Rightarrow \frac{\partial \ell_i}{\partial w} = (p_i - \tilde{y}_i) x_i,$$

since $z_i = w^\top x_i + b$ and $\frac{\partial z_i}{\partial w} = x_i$. Then, we note that

$$\frac{\partial \ell_i}{\partial b} = p_i - \tilde{y}_i,$$

since $\frac{\partial z_i}{\partial b} = 1$. So, the gradients of the regularized empirical risk loss function is given by

$$\nabla_w \mathcal{L}(w, b) = \frac{1}{n} \sum_{i=1}^{n} (p_i - \tilde{y}_i) x_i + w, \quad \frac{\partial \mathcal{L}}{\partial b} = \frac{1}{n} \sum_{i=1}^{n} (p_i - \tilde{y}_i) + b.$$

For the first optimization algorithm, which is the full-batch gradient descent method, we start from an initial guess (w_0, b_0), and then for $t = 0, 1, \ldots, T-1$, where T is the total number of full-gradient steps (iterations) that we choose to perform, we iterate by doing

$$w_{t+1} = w_t - \gamma_t \nabla_w \mathcal{L}(w_t, b_t), \quad b_{t+1} = b_t - \gamma_t \frac{\partial \mathcal{L}}{\partial b}(w_t, b_t),$$

and since we have a constant learning rate, then $\gamma_t = \gamma = 0.001$.

For the second optimization algorithm, which is the mini-batch stochastic gradient descent (mini-batch SGD) method, we first let $b = 10$ be the mini-batch size. At each iteration t, we sample a subset $B \subseteq S$ with $|B| = b$, and form the stochastic gradients

$$g_t^w = \frac{1}{b} \sum_{i \in B} \nabla_w \mathcal{L}(w_t, b_t), \quad g_t^b = \frac{1}{b} \sum_{i \in B} \frac{\partial \mathcal{L}}{\partial b}.$$

For $t = 0, 1, \ldots, T-1$, we then iterate using the update rule given by

$$w_{t+1} = w_t - \gamma_t g_t^w, \quad b_{t+1} = b_t - \gamma_t g_t^b,$$

where we again have a constant learning rate $\gamma_t = \gamma = 0.001$.

To implement the two algorithms in Python, we first restrict both the training and test sets to only the digits 3 and 8 by building boolean masks and apply them to X and y. We then flatten each 28×28 image into a 784-dimensional vector, normalize pixel values to $[0, 1]$, and convert the filtered labels into binary targets ($8 \mapsto 1, 3 \mapsto 0$), as mentioned earlier. At this point, `X_train` and `X_test` each have shape (11982, 784).

Next, we define `sigmoid(z)` for the logistic link and `compute_loss_and_grad(w,b,X,y)`, which returns the regularized cross-entropy loss plus its gradients w.r.t. w and b. Inside this loss routine, we compute the score vector z, apply the sigmoid to get probabilities, evaluate the average cross-entropy, add an l_2 penalty, and derive `grad_w` and `grad_b` as stated above.

With our data and helper functions in place, we implement two training loops:

- In `train_gd`, we initialize w and b to zero and do 1000 iterations of full-batch gradient descent. At each step, we call `compute_loss_and_grad` on the entire training set, take a step of size $\gamma = 0.001$ in the negative gradient direction, and at every 10th iterations, we append both the latest training loss to our histories.

- In `train_sgd`, we follow the same pattern but replace the full gradient with a stochastic estimate over a random mini-batch of size $b = 10$. We sample $b = 10$ indices without replacement, compute the gradient on that batch (plus the l_2 term), update w and b, and again record the training loss every 10th steps.

For the $T = 1000$ iterations, we plot the training loss after every 10 iterations:

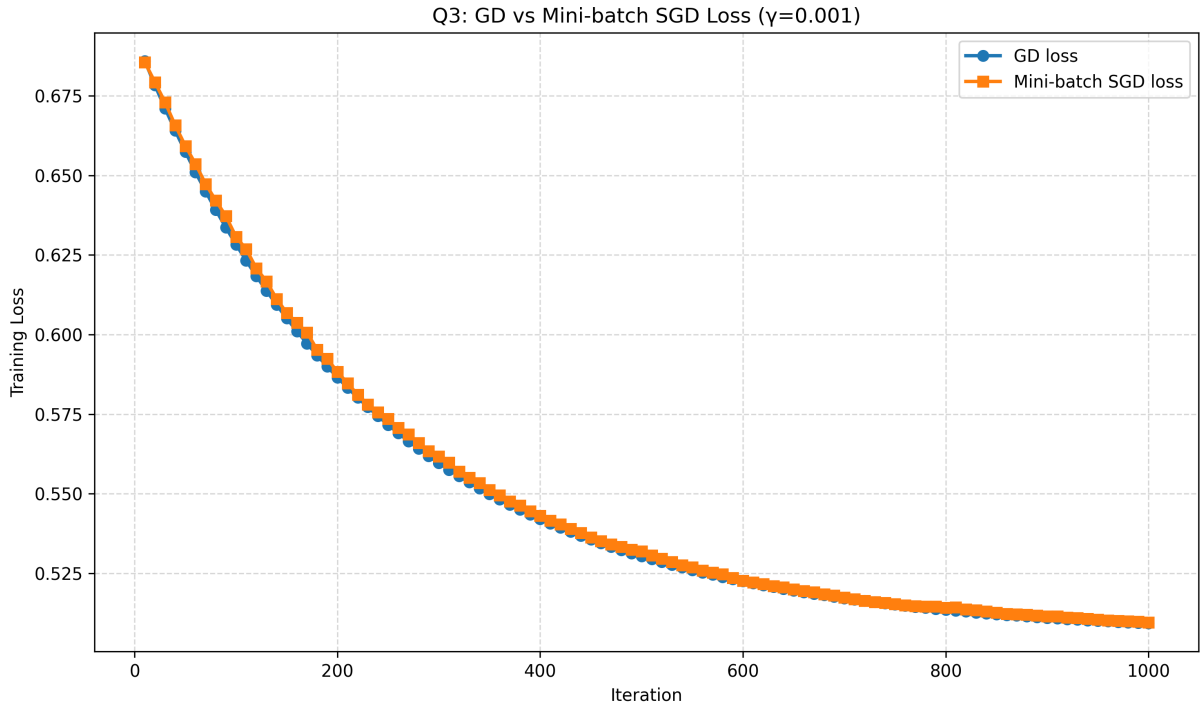

Figure 3: Comparison of full-batch gradient descent (GD) and mini-batch stochastic gradient descent (SGD with $b = 10$) on the regularized logistic-regression problem for MNIST digits $\{3, 8\}$.

In Figure 3, we plot the regularized cross-entropy loss on the entire training set after every 10 iterations for both the gradient descent (GD) and mini-batch stochastic gradient descent (SGD). Full-batch GD (blue) drops the loss in a quite perfectly smooth curve, while SGD (orange) follows almost the same downward trend but with small fluctuations, which are those come from the randomness of sampling just ten examples at a time. Both methods steadily reduce the loss toward approximately the same final value.

Response to Question 4. We have to show and explain how the training losses for the two different algorithms change with the learning rate γ, where we focus on $\gamma \in \{0.0001, 0.001, 0.01\}$. Thus, by reusing the code from Question 3, we looped the constant learning rate over $\gamma \in \{0.0001, 0.001, 0.01\}$ and recorded the training-loss curves (evaluated on the full training set every 10 iterations) for both full-batch gradient descent (GD) and mini-batch SGD (with batch size $b = 10$). The resulting plots are shown in Figure 4:

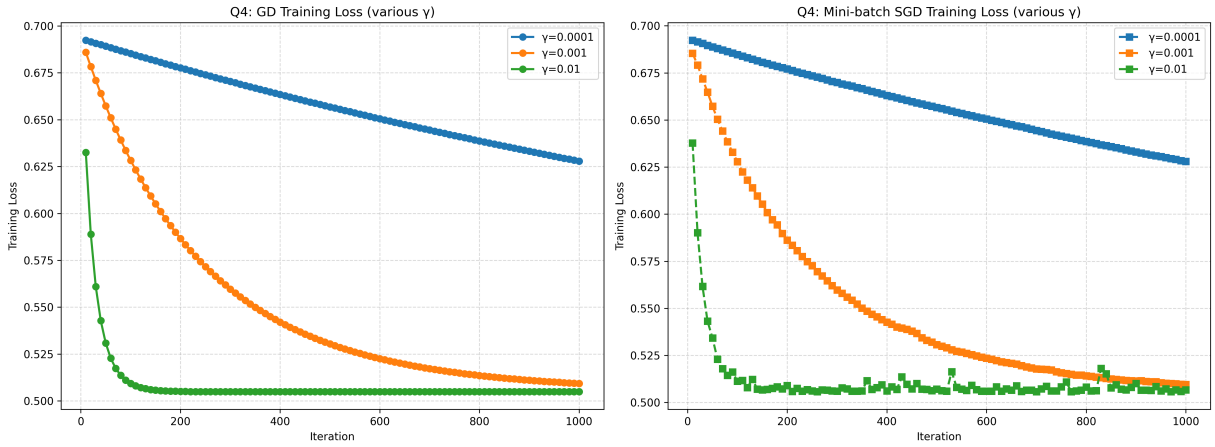

Figure 4: Comparison of training loss for full-batch gradient descent (GD) and mini-batch stochastic gradient descent (SGD with $b = 10$) on the regularized logistic-regression problem for MNIST digits $\{3, 8\}$ using different constant learning rates $\gamma \in \{0.0001, 0.001, 0.01\}$.

From the plots, we see that for the smallest rate $\gamma = 0.0001$, both algorithms make only very slow progress, where the loss after 1000 iterations remains above about 0.625, and the two curves are nearly identical, since such a tiny step dominates any minibatch variance. At $\gamma = 0.001$, the descent is much faster – GD decreases smoothly to about 0.52, and SGD follows closely with small oscillations due to sampling noise.

When we further increase the learning rate to $\gamma = 0.01$, we observe the steepest initial drop and lowest final losses (around 0.505 for both the GD and SGD algorithms). Although there are modest oscillations, neither algorithm diverges, and both settle at a lower final training-loss than with $\gamma = 0.001$. Consequently, we select $\gamma = 0.01$ as the optimal constant learning rate for both GD and mini-batch SGD, which we will use in the following questions.

Response to Question 5. We now focus on the role of batch size in the convergence behavior of mini-batch SGD. Building on our earlier experiments, we fix the constant learning rate at the optimal value $\gamma = 0.01$ (as determined in Question 4) and vary the batch size over $b \in \{1, 10, 100\}$. For each choice of b, we record the training loss on the full dataset every 10 iterations, then plot and compare the resulting curves in Figure 5:

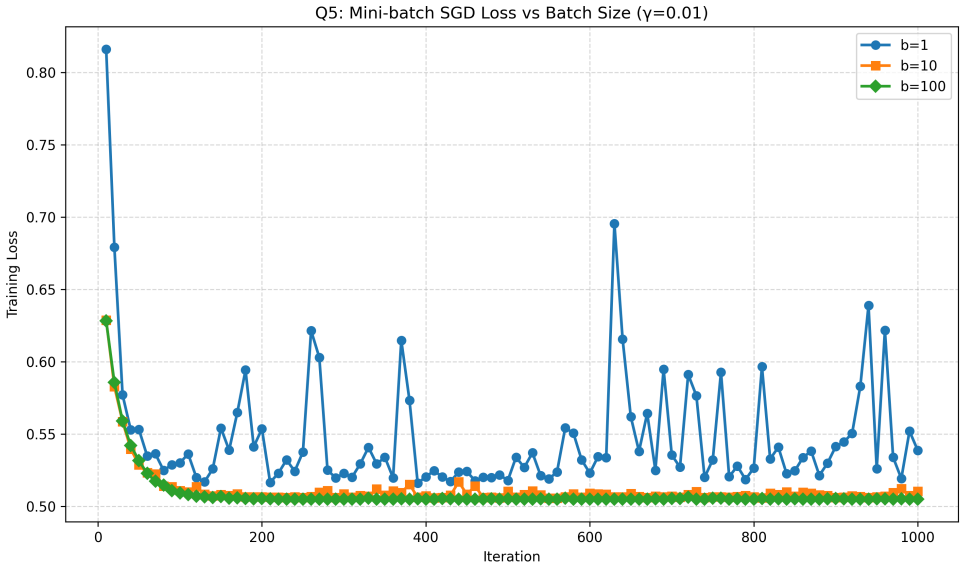

Figure 5: Mini-batch SGD training loss ($\gamma = 0.01$) for different batch sizes $b \in \{1, 10, 100\}$ on the regularized logistic-regression problem for MNIST digits {3,8}.

With a batch of size $b = 1$, the loss curve is quite jumpy, since we see huge upward spikes followed by sharp drops, and although it occasionally goes below 0.55, on average it drifts more slowly, only reaching around 0.525 after 1000 iterations. These huge fluctuations happen because each update is based on a single random example, so the gradient direction can change radically from one iteration to the next, causing the big spikes we see.

When we increase the batch size to $b = 10$, the training-loss drops quickly to about 0.505 with small, frequent fluctuations. This shows that averaging over ten examples cuts down most of the randomness, letting the algorithm move steadily toward the minimum. When using $b = 100$ examples per batch, it makes the curve almost perfectly smooth, reaching a similar final loss slightly faster in iteration count. However, since each of those steps costs ten times more work than $b = 10$, the smaller batch gives almost all of the benefit in far less computation per update, so $b = 10$ is a good compromise between speed and stability.

Response to Question 6. Now, we have to re-run the mini-batch SGD algorithm with batch-size $b = 10$ and a diminishing learning rate given by

$$\gamma_t = \frac{1}{t},$$

where $t = 1, 2, ...$ and see if it improves the performance over the best constant learning rate from Question 4 that we chose as $\gamma = 0.01$. Thus, by modifying the SGD loop from the previous questions to now use a diminishing step-size γ_t as stated above, and record the full-batch loss for every ten updates, we obtain the training losses plotted in Figure 6:

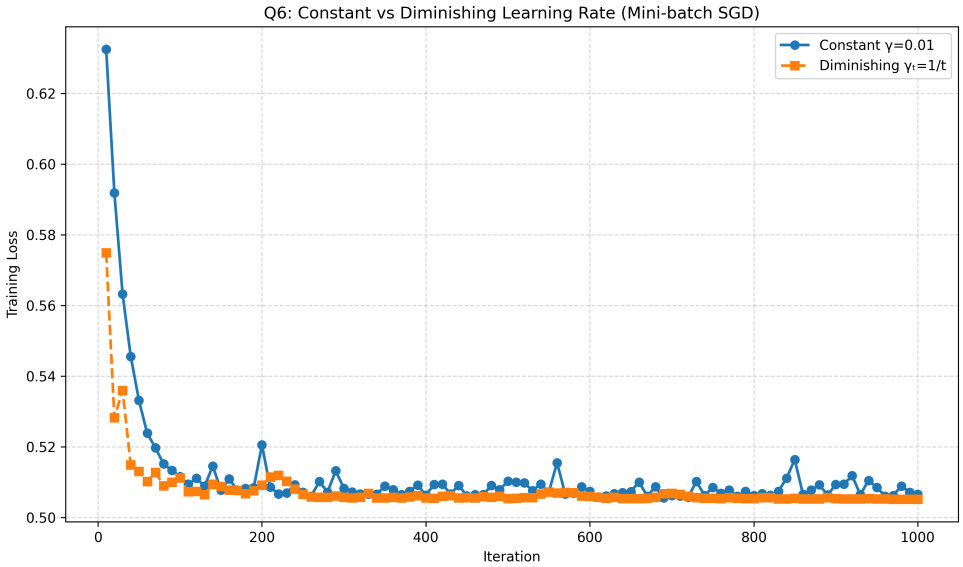

Figure 6: Comparison of SGD ($b = 10$) with a constant learning rate $\gamma = 0.01$ versus a diminishing learning rate $\gamma_t = 1/t$ on the regularized logistic-regression problem for MNIST digits {3,8}.

The orange curve shows SGD with the diminishing learning rate $\gamma_t = 1/t$, and the blue curve is the constant learning rate $\gamma = 0.01$. We see that the diminishing learning rate run drops more sharply in the very first few dozen iterations, since $\gamma_1 = 1, \gamma_2 = 0.5$ etc. give huge early steps that quickly push the loss down. After about 50–100 iterations, however, the learning rate has decayed so much that the descent slows and the curve flattens out.

By contrast, the constant learning rate run (blue) descends a bit more steadily. We see that it does not benefit from the explosive early jump, but it also does not flatten out so fast in later iterations. Both methods end up around the same final loss (approximately 0.505), with the diminishing learning rate curve smoother in the tail.