Machine Learning B (2025)
Home Assignment 3

Carsten Jørgensen, student ID: skj730

Contents

1 SVM with Kernels

In Figure 1 we see 20 non-linear samples that we want to classify using a Support Vector Machine (SVM) with different kernels.

For this question, I am using the Support Vector Classification (SVC) algorithm from scikit-learn [Ped+11].

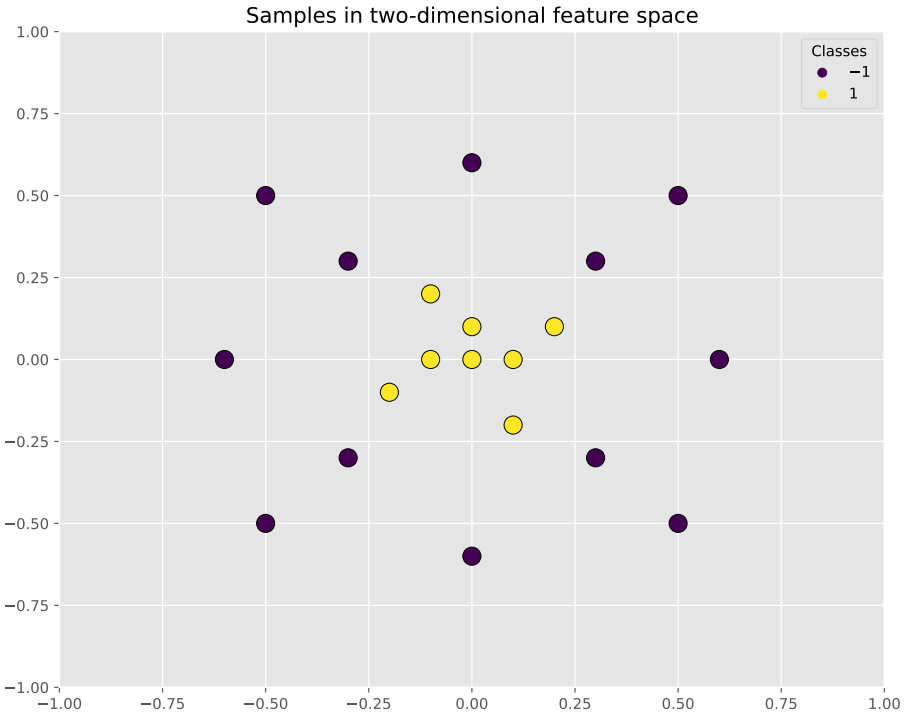

Figure 1: Non-linear samples in 2D

Response to Question 1.

For this question, we use a linear kernel with misclassification penalty $c = 1, 100, 1000$. See Figure 2 for a visualization of the result of the SVM algorithm. Tabel 1 reports the results.

In this case, no straight line can correctly separate a cluster of points that is completely surrounded by points of another class. This is a fundamental geometric limitation. The SVM optimization problem aims to minimize: $\frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i$ subject to constraints: $(w \cdot x_i + b) \geq 1 - \xi_i$ and $\xi_i \geq 0$.

C	Training Loss
1	8.0
100	8.0
1000	8.0

Table 1: Linear SVM Results Summary. A training loss of 8 indicates that all 8 point belonging to class 1 are misclassified.

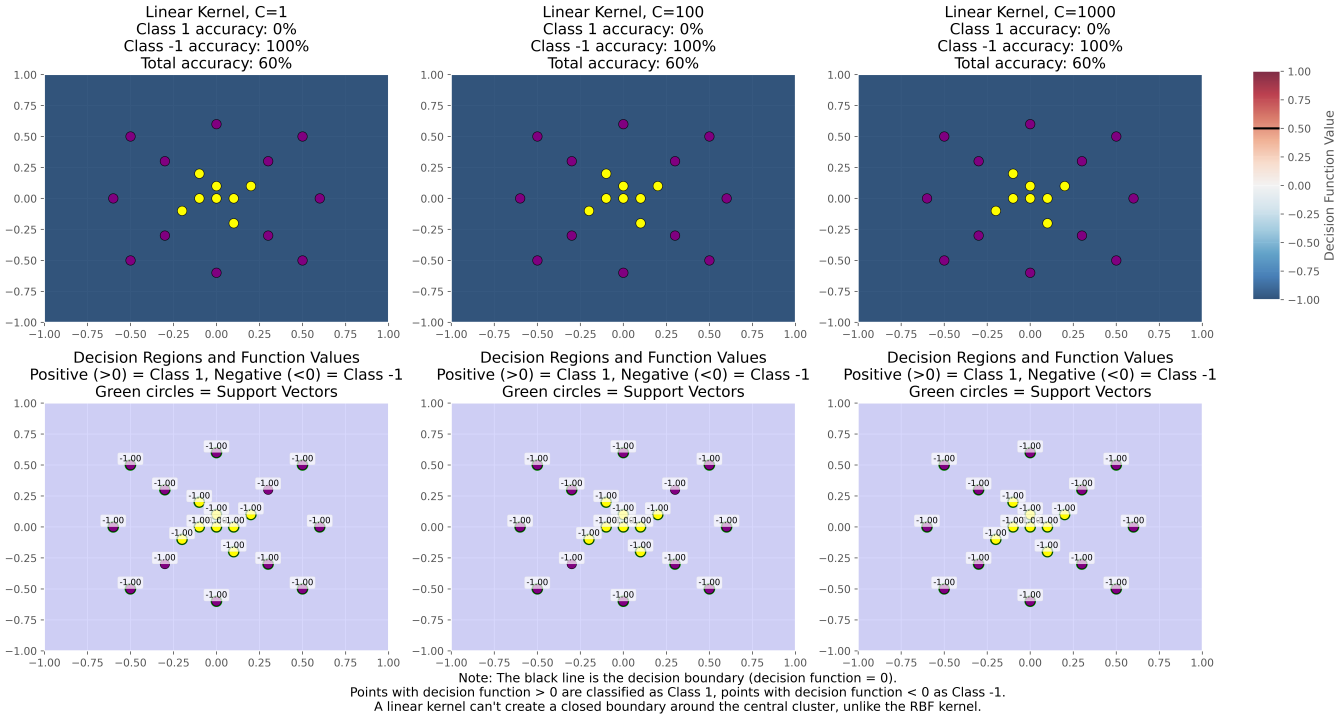

Figure 2: Visualization of decision boundary using a linear kernel

a	Training Loss
0.1	0.0
1	8.0
10	8.0

Table 2: RBF SVM Results Summary. A training loss of 8 indicates that all 8 point belonging to class 1 are misclassified.

With a linear kernel and encircled data:

1. The optimization finds that allowing all 8 class 1 points to be misclassified (with their corresponding slack variables ξ_i) results in a lower objective function value than any alternative linear boundary

2. The misclassification of all 8 points becomes the "optimal" solution to this mathematical problem

Response to Question 2.

With a non-linear kernel like the Gaussian kernel we are in one case able to correctly classify all 20 points. This is in the case where $a = 0.1$. Training losses are reported in Table 2.

In Figure 3 in the lower-middle plot, we see that SVM predicts ~ 0.70 for the point a class 1 when $a = 1$. The points are still misclassified, but not as clearly for the case when $a = 10$ (lower-right plots), where all the class 1 points get a decision value of -1.

When a is too high (1 or 10), the SVM model is likely to create a simple boundary that classifies most of the space as class -1 (the majority class). Without the ability to create a "hole" in the decision boundary, all the class 1 points are misclassified.

2 Logistic regression on MNIST

For this exercise, we want to classify digits from the MNIST data set as either 3 or 8 t. We will use two different optimization algorithms for the task. Gradient Descent and mini-batch Stochastic Gradient Descent. I have implemented both algorithms from scratch in order to gain practical experience about their implementation.

Class distribution

The data set consists of 13966 samples that are divided into a traning set containing 11172 samples (80%) and a test set with 2974 samples with the following distribution:

- Training set: 5713 instances of digit 3, 5459 instances of digit 8

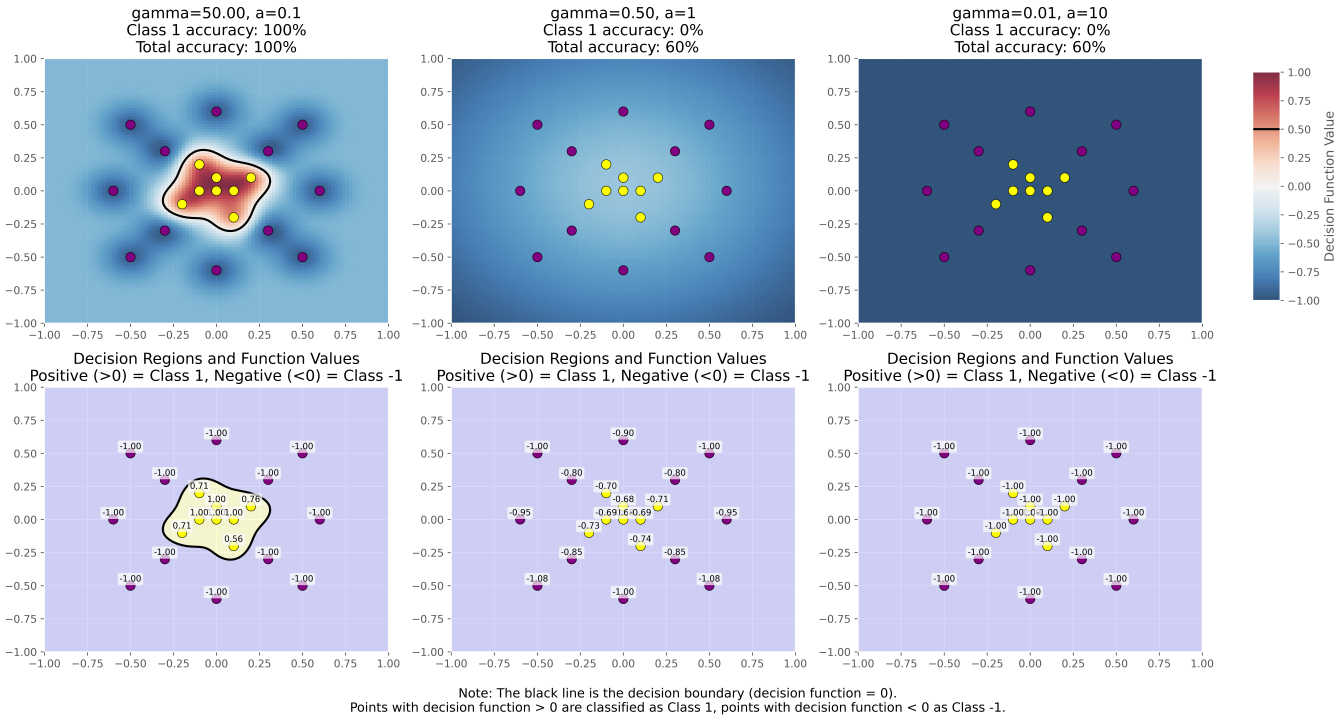

Figure 3: Visualization of decision boundary using a RBF kernel

- Test set: 1428 instances of digit 3, 1366 instances of digit 8

In Figure 4 we see some samples from the data set.

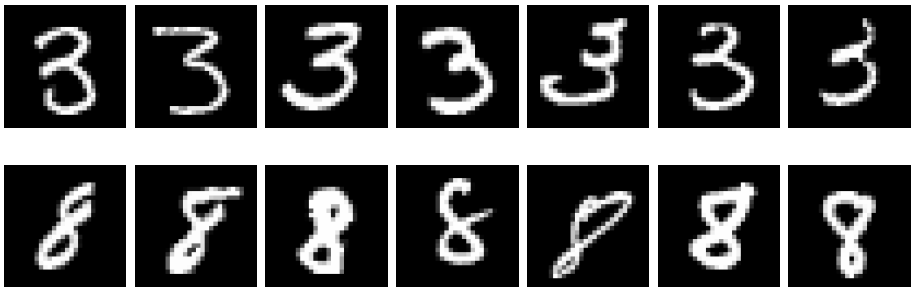

Figure 4: Sample images from data set

Problem setup

The setup is as following. Given a dataset $S = \Big\{(x_1, y_1), \ldots, (x_n, y_n)\Big\}$ where:

- $x_i \in \mathbb{R}^m$ ($m = 784$ for MNIST images flattened to vectors)
- $y_i \in \{0, 1\}$ (0 for digit 3, 1 for digit 8)

We define:

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

for weights $w \in \mathbb{R}^m$ and a bias term $b \in \mathbb{R}$. The loss function is cross-entropy with $L2$-regularization:

$$\mathcal{L}(w, b) = \frac{1}{n} \sum_{i=1}^{n} \ell_i(w, b) + \frac{\mu}{2} \Big(\|w\|^2 + b^2 \Big)$$

where $\ell_i(w, b) = -\Big[y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \Big]$

The gradients of the loss function are:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{n} \sum_{i=1}^{n} (p_i - y_i) x_i + \mu w$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{n} \sum_{i=1}^{n} (p_i - y_i) + \mu b$$

See Appendix A for a derivation of the gradients. Note that the implementation of the gradients do not use summation over single terms but are implemented as vectorized operations.
The implementation of gradients and loss function can be seen in the Appendix code listing 1 and 2. The core part of the training algorithm is listed here 3

Response to Question 3.

In Figure 5 we plot the training losses for the different algorithms for each epoch. Gradient descent converges very slow with a learning rate $\gamma = 0.001$, so here we have 1200 epochs. Mini-batch SGD converges faster but due to the stochastic element of this algorithm the training loss "jumps" up and down.

Response to Question 4.

For this question, we compare the impact of the learning rate γ on the two different algorithms. In Table 3 we see the final loss and test accuracy of the different combinations. Figures 6 and 7 show the training loss for each epoch for the GD resp. mini-batch SGD algorithms.

(a) Gradient Descent

(b) Mini-batch SGD

Figure 5: Gradient descent method with a constant learning rate $\gamma = 0.001$. Mini-batch stochastic gradient descent (mini-batch SGD) method with batch-size $b = 10$ and a constant learning rate $\gamma = 0.001$.

Method	Learning Rate	Batch Size	Final Loss	Test Accuracy
GD	0.0001	N/A	0.6540	0.8486
GD	0.001	N/A	0.5089	0.9102
GD	0.01	N/A	0.5030	0.9202
SGD	0.0001	10	0.5030	0.9198
SGD	0.001	10	0.5032	0.9234
SGD	0.01	10	0.5065	0.9166

Table 3: Summary of Gradient Descent (GD) and mini-batch SGD with different learning rates

Figure 6: Comparison of learning rate impact on GD

Figure 7: Comparison of learning rate impact on mini-batch SGD

Analysis of learning rate impact on GD

For μ-strongly convex functions[1], the gradient descent converges linearly at a rate of approximately $(1 - \eta\mu)^t$ where:

- γ is the learning rate
- $\mu = 1$ is the strong convexity parameter
- t is the number of iterations.

This is from the result of Theorem 3.6 in [GG24]. For our three cases:

1. Small learning rate ($\gamma = 0.0001$, red line):
    - Convergence factor: $(1 - 0.0001)^t$
    - The algorithm takes very small steps
    - Even after 1200 epochs, loss only reduces to ~ 0.65
    - Still far from convergence
2. Medium learning rate ($\gamma = 0.001$, blue line):
    - Convergence factor: $(1 - 0.001)^t$
    - Makes reasonable progress
    - After 1200 epochs, loss reaches ~ 0.51
    - Getting close to convergence
3. Higher learning rate ($\gamma = 0.01$, purple line):
    - Convergence factor: $(1 - 0.01)^t$
    - Converges rapidly
    - Reaches ~ 0.5 loss in just 200 epochs
    - Essentially converged

Analysis of learning rate impact on mini-batch SGD

For mini-batch SGD with a constant learning rate η on a μ-strongly convex function, the convergence rate is characterized by Theorem 6.11 in [GG24]. The theorem shows us that the converges rate is determined by 2 factors:

1. The factor $(1 - \eta\mu)^t$ is still present as in GD and can here be characterized as **Optimization error**

[1] We know from the lecture notes that the loss function for this problem is 1-strongly convex

2. We also have the term with is a **Statistical error** term due to gradient noise $\frac{2\gamma\sigma_b^*}{\mu}$, where σ_b^* is the minibatch gradient noise [GG24] remark 6.6.

For our three cases:

1. Small learning rate ($\gamma = 0.0001$, red line):
    - Much faster convergence that GD, approaching optimal within 50 epochs
    - While each step is small, mini-batch SGD makes $n/10$ updates per epoch versus just 1 in full-batch GD, resulting in dramatically faster progress per epoch.
2. Medium learning rate ($\gamma = 0.001$, blue line):
    - Fast convergence, reaching optimal in about 5 epochs with minimal oscillation
    - Provides an excellent balance between step size and stability for mini-batch
3. Higher learning rate ($\gamma = 0.01$, purple line):
    - Very rapid initial convergence but with noticeable oscillations throughout training
    - Larger learning rate leads to faster optimization error reduction but higher statistical error

Mini-batch SGD vs. GD

- **Update Frequency**: Mini-batch SGD makes $n/\text{batch\_size} = n/10$ parameter updates per epoch versus 1 for full-batch GD, and I believe this explains the dramatic reduction in required epochs.
- **Convergence Speed vs. Stability Trade-off**:
    - Small η (0.0001): Stable but slower convergence
    - Medium η (0.001): Excellent balance of speed and stability
    - Large η (0.01): Fastest initial progress but ongoing oscillations
- **Final Loss Convergence**: All learning rates eventually reach similar final loss values (around $\sim 0.5 - 0.51$), confirming they all approach the global minimum as expected for a strongly convex problem.
- **Oscillation Behavior**: The visible oscillations with $\eta = 0.01$ reflect the statistical error term in the convergence bound. These oscillations are a fundamental characteristic of mini-batch SGD with larger learning rates.

Response to Question 5.

From the previous question, we conclude that using mini-batch SGD for this specific $L2$-regularized logistic regression problem the optimal learning rate is $\gamma = 0.001$ as it provides the best performance, offering fast convergence with minimal oscillation.

But the analysis also highlights that while mini-batch SGD makes learning much more efficient in terms of epochs, careful learning rate selection remains critical for optimal performance.

Figure 8 shows the impact of the batch size on the training loss of the mini-batch SGD algorithm. We can use remark 6.6 in [GG24] to understand the plot. For larger batch sizes the behavior of mini-batch SGD converges to Gradient Descent, and hence we here see a smooth converges rate for the training loss. When batch size is 1, mini-batch SGD is identical to SGD. In Remark 6.6 of [GG24], the authors explicitly state that mini-batch SGD "interpolates between single and full batches" where:

- At $b = 1 : \sigma_b^* = \sigma_f^*$ (full gradient noise)
- At $b = n : \sigma_b^* = 0$ (no gradient noise)

From Theorem 5.7 and 6.11 in [GG24] we see that the only difference in convergence for SGD and mini-batch SGD are the factors σ_f^* and σ_b^*. From Lemma 6.5 we have the relationship

$$\sigma_b^* = \frac{n-b}{b(n-1)}\sigma_f^*$$

For a batch size b with $1 < b < n$ the factor $\frac{n-b}{b(n-1)}$ is less than 1 so σ_b^* must be less than σ_f^*. This decreasing gradient noise is a key factor in why mini-batch SGD achieves faster convergence in terms of iterations required than SGD. This is exactly what we observe in Figure 8.

Response to Question 6.

In Figure 9 we compare the traning loss for mini-batch SGD with constant learning rate 0.001 and a diminishing learning rate schedule $\frac{1}{t}, t = 1, 2, \ldots$. The test accuracy for the constant learning rate is 0.9234 whereas it is 0.9202 for the diminishing schedule. So overall the 2 approaches give more or less the same result.

We know from Theorem 6.11 in [GG24]that the term $\frac{2\gamma\sigma_b^*}{\mu}$ does not vanish as t increases, which mathematically demonstrates that constant learning rate SGD does not converge exactly to the optimum but maintains some error term related to the gradient noise. I guess that the diminishing learning rate changes that as it now goes to 0 such that the diminishing learning rate maintains a more consistent downward trajectory.

So from a stability point of view the diminishing learning rate (blue line) demonstrates a much smoother convergence pattern after the initial drop making it preferable to a constant learning rate.

Figure 8: Comparison of batch size impact on SGD

Figure 9: SGD with diminishing learning rate $\gamma = \frac{1}{t}, t = 1, 2, \ldots$

A Appendix

Derivation of Gradients

Here we derive the gradients for Logistic Regression with $L2$ Regularization step by step. The loss function is given by:

- $\mathcal{L}(w, b) = \frac{1}{n} \sum_{i=1}^{n} \ell_i(w, b) + \frac{\mu}{2}(\|w\|^2 + b^2)$
- Where $\ell_i(w, b) = -[y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$

Step 1: Calculate $\partial \ell_i / \partial p_i$. Starting with the cross-entropy term:

$$\ell_i(w, b) = -[y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

Taking the derivative with respect to p_i:

$$\begin{aligned} \frac{\partial \ell_i}{\partial p_i} &= -\left[\frac{y_i}{p_i} - \frac{1 - y_i}{1 - p_i}\right] \\ &= -\frac{y_i}{p_i} + \frac{1 - y_i}{1 - p_i} \end{aligned}$$

Finding a common denominator:

$$\begin{aligned} \frac{\partial \ell_i}{\partial p_i} &= \frac{(1 - y_i)p_i - y_i(1 - p_i)}{p_i(1 - p_i)} \\ &= \frac{p_i - y_i p_i - y_i + y_i p_i}{p_i(1 - p_i)} \\ &= \frac{p_i - y_i}{p_i(1 - p_i)} \end{aligned}$$

Step 2: Calculate $\partial p_i / \partial z_i$. For the sigmoid function $p_i = \frac{1}{1+\exp(-z_i)}$, the derivative is:

$$\frac{\partial p_i}{\partial z_i} = \frac{e^{-z_i}}{(1 + e^{-z_i})^2} = p_i(1 - p_i)$$

This can be verified by:

$$\begin{aligned} p_i(1 - p_i) &= \frac{1}{1 + e^{-z_i}} \times \left[1 - \frac{1}{1 + e^{-z_i}}\right] \\ &= \frac{1}{1 + e^{-z_i}} \times \frac{e^{-z_i}}{1 + e^{-z_i}} \\ &= \frac{e^{-z_i}}{(1 + e^{-z_i})^2} \end{aligned}$$

Step 3: Calculate $\partial z_i/\partial w$ and $\partial z_i/\partial b$. For $z_i = w^T x_i + b$:

$$\frac{\partial z_i}{\partial w} = x_i \quad \text{(this is a vector)}$$
$$\frac{\partial z_i}{\partial b} = 1$$

Step 4: Apply the Chain Rule to Find $\partial \ell_i/\partial w$ and $\partial \ell_i/\partial b$.

$$\begin{aligned}\frac{\partial \ell_i}{\partial w} &= \frac{\partial \ell_i}{\partial p_i} \times \frac{\partial p_i}{\partial z_i} \times \frac{\partial z_i}{\partial w} \\ &= \frac{p_i - y_i}{p_i(1-p_i)} \times p_i(1-p_i) \times x_i \\ &= (p_i - y_i) \times x_i\end{aligned}$$

$$\begin{aligned}\frac{\partial \ell_i}{\partial b} &= \frac{\partial \ell_i}{\partial p_i} \times \frac{\partial p_i}{\partial z_i} \times \frac{\partial z_i}{\partial b} \\ &= \frac{p_i - y_i}{p_i(1-p_i)} \times p_i(1-p_i) \times 1 \\ &= p_i - y_i\end{aligned}$$

Step 5: Calculate the Complete Gradients Including Regularization. For the full loss function $\mathcal{L}(w, b) = \frac{1}{n}\sum_{i=1}^{n} \ell_i(w, b) + \frac{\mu}{2}(\|w\|^2 + b^2)$. The derivative of the regularization term:

$$\frac{\partial(\|w\|^2)}{\partial w} = 2w$$
$$\frac{\partial(b^2)}{\partial b} = 2b$$

Therefore, the complete gradients are:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{n}\sum_{i=1}^{n}(p_i - y_i)x_i + \mu w$$
$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{n}\sum_{i=1}^{n}(p_i - y_i) + \mu b$$

Code listings

```
def compute_gradients(X, y, w, b, mu):
    """
    Compute the gradients of the loss function with respect to parameters
    .

    Args:
        X: Features
        y: True labels
        w: Weights
        b: Bias
        mu: Regularization parameter

    Returns:
        dw: Gradient with respect to w
        db: Gradient with respect to b
    """
    n = len(y)
    p = predict(X, w, b)

    # Gradient for w
    dw = np.dot(X.T, (p - y)) / n + mu * w

    # Gradient for b
    db = np.sum(p - y) / n + mu * b

    return dw, db
```

Listing 1: Gradients

```
def predict(X, w, b):
    """
    Make predictions using the logistic regression model.

    Args:
        X: Features
        w: Weights
        b: Bias

    Returns:
        Predicted probabilities
    """
    z = np.dot(X, w) + b
    return sigmoid(z)

def compute_loss(X, y, w, b, mu):
    """
    Compute the cross-entropy loss with L2 regularization.

    Args:
        X: Features
        y: True labels
```

```
        w: Weights
        b: Bias
        mu: Regularization parameter

    Returns:
        Total loss value
    """
    p = predict(X, w, b)

    # Avoid log(0) errors
    p = np.clip(p, 1e-15, 1 - 1e-15)

    # Cross-entropy loss
    cross_entropy = -np.mean(y * np.log(p) + (1 - y) * np.log(1 - p))

    # L2 regularization
    regularization = (mu / 2) * (np.sum(w**2) + b**2)

    return cross_entropy + regularization
```

Listing 2: Loss function

```
def train_model(
    X_train,
    y_train,
    learning_rate,
    num_epochs,
    mu,
    method="vanilla",
    batch_size=10,
    diminishing_lr=False,
    verbose=True,
):
    """
    Train a logistic regression model using either vanilla gradient
   descent or mini-batch SGD.

    Args:
        X_train: Training features
        y_train: Training labels
        learning_rate: Initial learning rate for gradient updates
        num_epochs: Number of training epochs
        mu: Regularization parameter
        method: 'vanilla' for vanilla gradient descent or 'sgd' for mini-
   batch SGD
        batch_size: Size of mini-batches (only used if method='sgd')
        diminishing_lr: If True, use diminishing learning rate gamma =
   learning_rate/t for SGD
        verbose: Whether to print progress

    Returns:
        w: Trained weights
```

```
    b: Trained bias
    loss_history: History of loss values during training
"""
m = X_train.shape[1]  # Number of features (784 for MNIST)
n = X_train.shape[0]  # Number of training examples

# Initialize parameters
w, b = initialize_parameters(m)

# Initialize loss history
loss_history = []

# Loop over epochs
for epoch in range(num_epochs):
    # Calculate current learning rate for SGD if using diminishing
schedule
    current_lr = learning_rate
    if method == "sgd" and diminishing_lr:
        # t starts from 1, so epoch + 1
        current_lr = learning_rate / (epoch + 1)
        if verbose and (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch + 1}: Current learning rate = {
current_lr:.6f}")

    if method == "vanilla":
        # Vanilla gradient descent: compute gradients on entire
dataset
        dw, db = compute_gradients(X_train, y_train, w, b, mu)

        # Update parameters
        w = w - learning_rate * dw
        b = b - learning_rate * db

    elif method == "sgd":
        # Mini-batch SGD: process data in batches

        # Shuffle data
        indices = np.random.permutation(n)
        X_shuffled = X_train[indices]
        y_shuffled = y_train[indices]

        # Process each mini-batch
        for i in range(0, n, batch_size):
            X_batch = X_shuffled[i : i + batch_size]
            y_batch = y_shuffled[i : i + batch_size]

            # Compute gradients on mini-batch
            dw, db = compute_gradients(X_batch, y_batch, w, b, mu)

            # Update parameters with current learning rate
            w = w - current_lr * dw
```

```
                b = b - current_lr * db
        else:
            raise ValueError("Method must be either 'vanilla' or 'sgd'")

        # Compute and store loss for the epoch
        loss = compute_loss(X_train, y_train, w, b, mu)
        loss_history.append(loss)

        # Print loss every 10 epochs as required
        if verbose and (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss:.4f}")

    return w, b, loss_history
```

Listing 3: Training algorithm

References

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Rud17] Sebastian Ruder. *An Overview of Gradient Descent Optimization Algorithms*. June 15, 2017. DOI: 10.48550/arXiv.1609.04747. arXiv: 1609.04747 [cs]. URL: http://arxiv.org/abs/1609.04747 (visited on 05/12/2025). Pre-published.

[Mur22] Kevin P. Murphy. *Probabilistic Machine Learning: An Introduction*. Adaptive Computation and Machine Learning. Cambridge, Massachusetts London, England: The MIT Press, 2022. 826 pp. ISBN: 978-0-262-36930-5 978-0-262-04682-4. URL: https://probml.github.io/pml-book/book1.html.

[GG24] Guillaume Garrigos and Robert M. Gower. *Handbook of Convergence Theorems for (Stochastic) Gradient Methods*. Mar. 9, 2024. DOI: 10.48550/arXiv.2301.11235. arXiv: 2301.11235 [math]. URL: http://arxiv.org/abs/2301.11235 (visited on 04/25/2025). Pre-published.