NUMERICAL OPTIMISATION
PROJECT GUIDELINES

Optimisation in classification with Support Vector Machines (SVMs)

Marta Betcke

Submit **a single PDF report** via Moodle by **4pm (UK time) on Wednesday 23rd of April 2025**.

To facilitate marking, please **include the optimisation method in the name of the submitted PDF file** and in the header/title of your report.

Scope:
This is a numerical optimisation and not a machine learning project. The emphasis here is on **optimisation methods and their convergence** not on classification performance. The task is to take an application problem, translate it into mathematical optimisation formulation abstract from the application, then solve and analyse it as an optimisation problem.

Report:
To achieve full marks (as in brackets **[n pt]**) the report needs to be well structured (address each point completely in a dedicated section), be written clearly and logically while succinct, mathematically rigorous (on par with the lecture), use consistent notation, and be self contained i.e. include everything the reader needs to understand it. This does not mean you should include proofs of theorems, find a way to make a coherent argument without including the proofs.

Always provide references to sources of any claims, theorems, methods etc (avoid Wikipedia, good Wiki contributions state original sources!).

Do not expect us to search in your text for arguments, guess what you may have meant, look things up, or accept any statements without coherent supporting argument with reference to sources.

3k words is a total hard limit (code excluded), please include the word count.

Code:
The optimisation method needs to be implemented by you from scratch, no use of optimisation packages is allowed, except for reusing codes made available by us in tutorials. For setting up the classification problem you can use dedicated packages, but do not use the built in optimisation routines. You can use Python or Julia but you may need to reimplement some routines that were provided in Matlab.

Include your implementation of the optimisation method and an excerpt from your testing script showing how it is called (keep it to minimum necessary to validate that you used your own implementation, use highlighting if necessary), as two separate and clearly labelled appendices in your report.

Collate report and code as one single PDF document for submission. The code will not count towards the word limit. **You will only get full points for parts II & III if code is included.**

UCL rules and regulations on late submission and plagiarism apply. Use of chatGPT and similar is not allowed.

Mathematical formulation of an SVM classification problem:

Support vector machines (SVMs) are a well established and rigorously founded technique for solution of classification problems in machine learning.

(a) Introduce a classification problem, e.g. linearly separate two sets of points in a plane. Visualise if possible. **[5 pt]**

(b) Choose an SVM formulation for your problem (e.g. primal/dual, linear/nonlinear, choice of loss etc) and justify your choice. Here, the task is to translate an application problem, a.k.a. your SVM classification problem, into the language of mathematical optimisation. Explain your mathematical formulation without including any ML derivations. **[10 pt]**

(c) Identify the type and properties of the resulting optimisation problem and any challenges it poses. **[5 pt]**

Marks: 20 pt baseline **[: 20pt]**

Optimisation method and convergence theory in the context of your problem:

Note: Theory will only be given full marks if there is a serious attempt on numerical solution.

(a) Propose an appropriate method for solution of your optimisation problem. Justify your choice w.r.t. applicability and efficiency: convergence type are rate for your problem, complexity, memory usage, etc. **[5 pt]**

(b) Describe the method in sufficient detail so it is clear how it works (state your sources, unify notation from different sources). **[10 pt]**

You may want to introduce a method which was not on the syllabus and apply it to your optimisation problem. The bonus points awarded will depend on the level of difficulty of the chosen method and the theory involved.

(c) Discuss if local or global convergence can be expected for your problem. For more details see below (†). **[5 pt]**

(d) Discuss theoretical convergence rates predicted for your problem. For more details see below (†). **[5 pt]**

(†) To argue convergence in (c,d) paraphrase theorems from the lecture or other respectable sources (books, journals, trusted lecture notes etc). Always state the complete result, this may involve combining multiple lemmas and theorems into a coherent theorem (reference all sources, unify notation!). Explain why this result applies to your problem i.e. check the assumptions against the properties of your problem. If you cannot exactly match the problem with the theory explain this clearly and discuss what is the departure from the theory in your case and what effect on convergence do you expect and why.
Marks: 25 pt baseline + up to **25 pt** bonus (method substantially different to those on the syllabus, different convergence theorems, etc). **[: 25 - 50 pt]**

Solution of the optimisation problem and discussion of the results:

Note: Without a serious attempt on numerical solution, the project will not achieve a pass mark.

(a) Solve the optimisation problem and visualise, if possible, or find another way to present your solution (this could be application specific). List relevant parameters and other choices you made (e.g. which line search was used, etc). Critically discuss the obtained solution. **[15 pt]**

(b) Provide one relevant convergence plot and discuss the empirical convergence rate qualitatively (e.g. linear/quadratic/superlinear, etc; no need to calculate the precise rate). **[5 pt]**

(c) Discuss the theoretical versus empirical convergence rates including all relevant checks, e.g. was trust region active or not, was step size 1 attained or not, etc. If there are any discrepancies between the two, explain possible reasons. **[5 pt]**

(d) Discuss the theoretical versus empirical performance of the method in terms of complexity, CPU time, memory used. **[5 pt]**

Marks: 30 pt baseline **[: 30 pt]**

NUMERICAL OPTIMISATION PROJECT

Fai Aldoussri
April 2025

Table of Contents:

Table of Figures

Mathematical formulation of an SVM classification problem

To formulate the SVM classification problem, we start with a simple binary classification task in $\mathbf{R}^2$. Each data point belongs to one of two classes, labeled +1 or –1 and we assume that the data is linearly separable, meaning that there is at least one straight line that can perfectly separate the two classes. [1] We define the dataset as:

$$\{(x_i, y_i)\}_{i=1}^{N}, \quad x_i \in \mathbb{R}^2, \quad y_i \in \{-1, +1\}$$

To keep the setup simple and easy to visualize, we define two small sets of points manually:

- **Class +1**:(3, 9), (4, 8.5), (5, 9.2), (6.2, 7), (4, 9.5), (2.5, 10), (3.5, 9.7), (5.5, 8.8), (4.8, 9.3)

- **Class –1**: (7, 3), (8, 2), (9, 2.5), (6.8, 6), (8.5, 1.2), (9.5, 1.7), (7.8, 2.3), (8.8, 1.5), (9.2, 2.1)

These points are chosen so that they can clearly be separated by a straight line, but I included a few points close to the decision boundary to ensure the hinge loss is activated. Figure 1 demonstrates the distribution of the points.

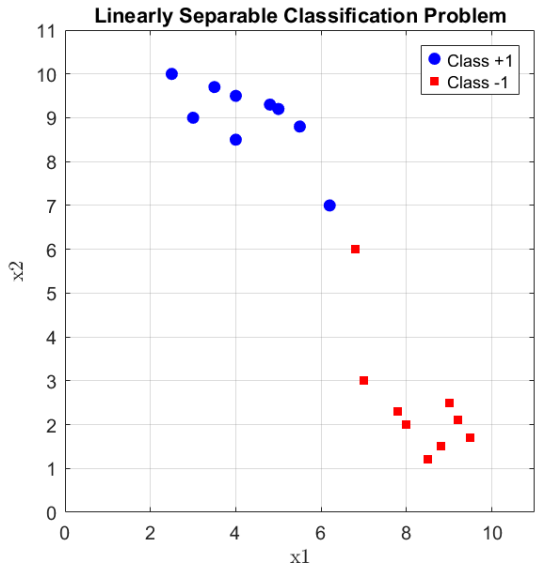

Figure 1: A linearly separable classification problem

For this problem, we choose the primal formulation. This choice is motivated by the small number of features relative to the number of training points, which makes the primal more computationally efficient than solving the dual, as discussed in. [1] The optimization problem has a compact structure: a quadratic regularization term combined with a hinge loss penalty, both expressed directly in the objective, without the need for explicit inequality constraints, which allows for efficient application of standard numerical methods. [2] We opt for a linear

SVM given that the dataset is linearly separable in its original two-dimensional feature space, without requiring a nonlinear kernel transformation [3]. Furthermore, to allow for margin violations in a mathematically tractable way and to improve numerical stability, we incorporate the hinge loss, resulting in a soft-margin formulation. The hinge loss can also be interpreted as solving a robust optimization problem that accounts for small variations in the input data. [4]

Let the objective function be:

$$f(w,b) := \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \max\left(0,\ 1 - y_i(w^\top x_i + b)\right)$$

We then consider the unconstrained optimisation problem:

$$\min_{w\in\mathbb{R}^2,\, b\in\mathbb{R}} \quad f(w,b)$$

Where:

$(w \in \mathbb{R}^2)$: decision variable representing the weight vector.

$(b \in \mathbb{R})$: decision variable representing the bias term.

$(x_i \in \mathbb{R}^2)$: input vector for data point i

$(y_i \in \{-1, +1\})$: class label for data point i

$(C > 0)$: regularisation parameter

$(\max(0,\ 1 - y_i(w^\top x_i + b)))$: hinge loss penalty for data point i

The formulation follows the soft-margin support vector machine [1]. This optimization problem is an unconstrained, convex, and non smooth problem; also, the decision variables are the vector w $\in$ R² and the scalar b $\in$ R, so the problem is defined over R³. It is composed of two components, a regularisation term and a penalty term. And because both of these components are convex functions, and the sum of convex functions remains convex. The first component $(\frac{1}{2}\|w\|^2)$ is a convex quadratic function, it is smooth and has a well-defined gradient. The second component $(\sum_{i=1}^{N} \max(0,\ 1 - y_i w^\top x_i + b))$ is the hinge loss which is convex but non smooth. Furthermore, there are no equality or inequality constraints on the variables, so this is an unconstrained problem. The resulting structure: a convex, nonsmooth objective without constraints is well known in convex optimization and falls within the class of problems amenable to first-order methods for nonsmooth optimization. [2]

Because the problem is convex, any local minimiser is also a global minimiser. However, the presence of non-smooth terms such as the hinge loss can pose numerical challenges. In particular, the lack of differentiability at certain points may limit the applicability of standard methods that rely on smoothness, and can affect the convergence behaviour of the optimisation process, for example, the convergence might be slower.

Optimisation method and convergence theory

I propose using the **Subgradient** method to solve the unconstrained, convex, non-smooth SVM formulation defined earlier, because the objective function includes a hinge loss term, which is convex but not differentiable, and although this lack of smoothness prevents the use of standard gradient based methods, subgradients can still be computed at all points, making the method applicable and effective in this context.[5] Moreover, as long as the iterates remain in a bounded region, the subgradients remain bounded, and the sublinear convergence bound applies. In this setting, the subgradient method achieves a **convergence rate** of O(1/√k) (where k is the number of iteration) when using a diminishing step size rule such as α_k = c / √k. Each iteration is computationally light, as it only involves evaluating the hinge loss across all data points and summing their contributions to compute the subgradient, resulting in a **total cost** of O(N) per iteration. In addition to its low computational cost, the method is also memory efficient, since it requires storing only the current iterate (w,b) and the associated subgradient, no Hessians or gradients from previous steps need to be stored. [5]

We minimize the function:

$$f(w,b) := \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \max\left(0,\ 1 - y_i(w^\top x_i + b)\right)$$

Let x = (w, b) $\in \mathbb{R}^3$ be the stacked variable vector.

The **subgradient** method is used to solve our convex, non-smooth SVM formulation. A **subgradient** of a convex function $f: \mathbb{R}^n \to \mathbb{R}$ at a point $x \in \mathbb{R}^n$ is any vector $g \in \mathbb{R}^n$ such that:

$$f(z) \geq f(x) + g^T(z - x), \quad \forall z \in \mathbb{R}^n$$

This inequality defines the set of all subgradients of **f** at **x** known as the subdifferential, **∂f(x)**.[6] At each iteration **k**, the method updates the iterate as:

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}, \quad \text{with } g^{(k)} \in \partial f(x^{(k)})$$

and $\alpha_k > 0$ is the step size.

We solve the SVM problem by updating the decision variables w $\in \mathbb{R}^2$ and b $\in \mathbb{R}$ using subgradients of the objective function. At each iteration **k**, a diminishing step size α_k = c / √k is used to ensure convergence for convex, non-smooth problems. [5]
The subgradient is computed using the set of margin violations:

$$I_k = \{i \mid y_i(w^\top x_i + b) < 1\}$$

Then the subgradients of the objective w.r.t **w** and **b** are then given by:

$$g_w = w - C\sum_{i\in\mathcal{I}_k} y_i x_i, \quad g_b = -C\sum_{i\in\mathcal{I}_k} y_i.$$

The update rule is:

$$w^{(k+1)} = w^{(k)} - \alpha_k g_w, \quad b^{(k+1)} = b^{(k)} - \alpha_k g_b.$$

The subgradient method is not a descent method, so we track the best objective value seen so far:

$$f_{\text{best}}^{(k)} = \min_{i\le k} f(x^{(i)}),$$

The choice of the diminishing step size is motivated by its robustness and theoretical convergence guarantees. Unlike constant step sizes, which may fail to converge, or Polyak's step size which depends on knowing the optimal objective value and can be unstable when estimated. This ensures convergence without requiring prior knowledge of the optimal value. Moreover, The method was implemented from scratch in MATLAB; see **Appendix A** for the full code.

I anticipate global convergence for the problem as the objective function is convex and consists of a smooth quadratic regularization term alongside a nonsmooth hinge loss. Under standard assumptions, convexity, local Lipschitz continuity, and a diminishing step size sequence satisfying $\sum \alpha_k = \infty, \quad \sum \alpha_k^2 < \infty$, the subgradient method guarantees convergence to the global minimum in function value at the rate O(1 / √k).[7] [8] While the full objective is not globally Lipschitz due to the unbounded quadratic term $\frac{1}{2}\|w\|^2$, the hinge loss component is Lipschitz continuous, and the presence of a coercive regularization term ensures that the objective grows as $\|w\|$ increases, which prevents the iterates from diverging. [8] Therefore, as long as the diminishing step size continues to shrink, the iterates remain within a bounded region. Consequently, this containment implies that within the region explored by the iterates, the function behaves like a Lipschitz function; more precisely, the subgradients remain bounded, satisfying local conditions for convergence. As a result, the assumptions required by the classical subgradient convergence theorems are met. Because the subgradient method is not a descent method, it is standard practice to track the best point found so far, that is, the one yielding the lowest function value.[5]
The combination of convexity and the step size condition leads to the result:

$$\lim_{k\to\infty} \min_{1\le i\le k} f(x^{(i)}) = \min_x f(x)$$

I predict that the theoretical convergence rate for this problem is to be sublinear, as is typical for subgradient methods applied to convex, non-smooth objectives. Specifically, under the standard assumptions of convexity, bounded subgradients, and a diminishing step size, the best function value achieved after k iterations satisfies the following inequality [5]:

$$f_{\text{best}}^{(k)} - f^\star \le \frac{R^2 + G^2 \sum_{i=1}^k \alpha_i^2}{2\sum_{i=1}^k \alpha_i},$$

Where:

- $f^\star$ is the global minimum of the objective function,
- $f_{\text{best}}^{(k)} = \min_{1\le i\le k} f(x^{(i)})$,
- R is the distance from the initial point to the optimal solution,
- G is an upper bound on the norm of subgradients,
- α_k is the step size used at iteration k.

This bound implies a sublinear convergence rate of:

$$f_{\text{best}}^{(k)} - f^* = \mathcal{O}\left(\frac{1}{\sqrt{k}}\right),$$

When diminishing step size rule $\alpha_k = \frac{c}{\sqrt{k}},$ is considered, which is widely used for ensuring convergence in convex, non-smooth optimization problems. This choice satisfies $\sum \alpha_k = \infty$ and $\sum \alpha_k^2 < \infty$, conditions necessary for the classical subgradient convergence theory to apply. Hence, all theoretical conditions for the classical subgradient convergence theory are satisfied, and I therefore expect sublinear convergence in function value.

Solution of the optimisation problem and discussion of the results

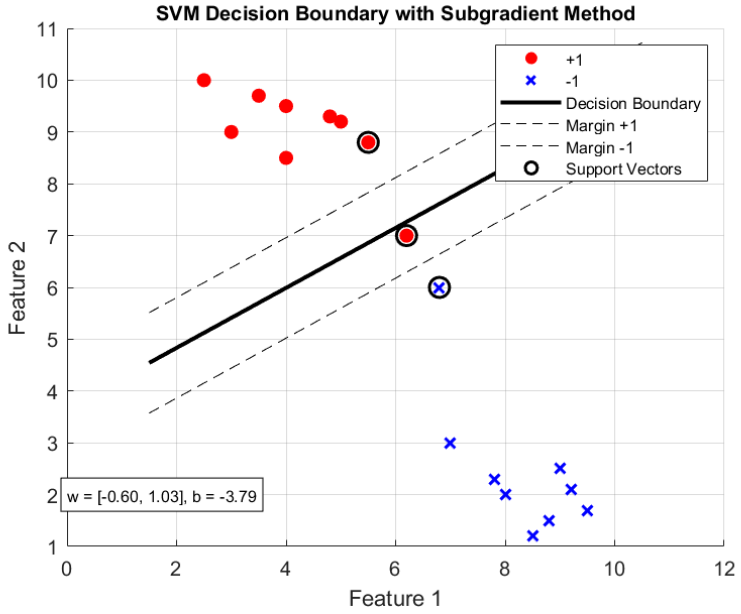

Figure 2: Solution of the Optimisation problem

Figure 2 presents the solution for the optimisation problem. The full implementation of the subgradient method to solve our objective function can be found on **Appendix A** and the key design choices used are as follows:

- Step size: I chose diminishing step size $\alpha_k = \frac{1}{\sqrt{k}}$ which satisfies the standard condition $\sum_{k=1}^{\infty} \alpha_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$ required by classical subgradient convergence theory for convex problems [5].
- Stopping criterion: $\|\nabla f(x_k)\| < 10^{-4}$ a threshold that ensures reasonably small subgradients without requiring excessive computation.
- Regularization parameter: C=1 this was selected as a design choice to balance margin maximization and classification flexibility.
- Maximum iterations: 100 chosen to allow sufficient progress while keeping runtime low.
- Initialization: $w = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad b = 0$ standard practice.

The use of line search or trust region strategies weren't included as these methods assume smoothness, which the hinge loss violates. Instead, the reliance was on classical subgradient convergence principles [5].

Overall, the subgradient method gave a reasonable separating hyperplane, and the decision boundary seems to classify the data correctly. The support vectors were found close to the margin, which is exactly what we'd expect from a soft-margin SVM. That said, the method has a few drawbacks. Most importantly, the final solution is entirely shaped by just a few support vectors. This is something that's well known in SVMs (only the points that sit on or inside the margin actually influence the model). Because of that, the boundary can be quite sensitive to the exact position of those points, especially when we're working with a small dataset and don't use any regularization beyond the hinge loss [8]. On top of that, subgradient methods don't follow a decent direction and use a fixed step size, so there's no guarantee that the last iterate we get is anywhere near optimal. From the plot, the margin looks a bit narrow and the boundary cautious, which might reflect the small number of active constraints or even some sensitivity to the way the data is distributed.

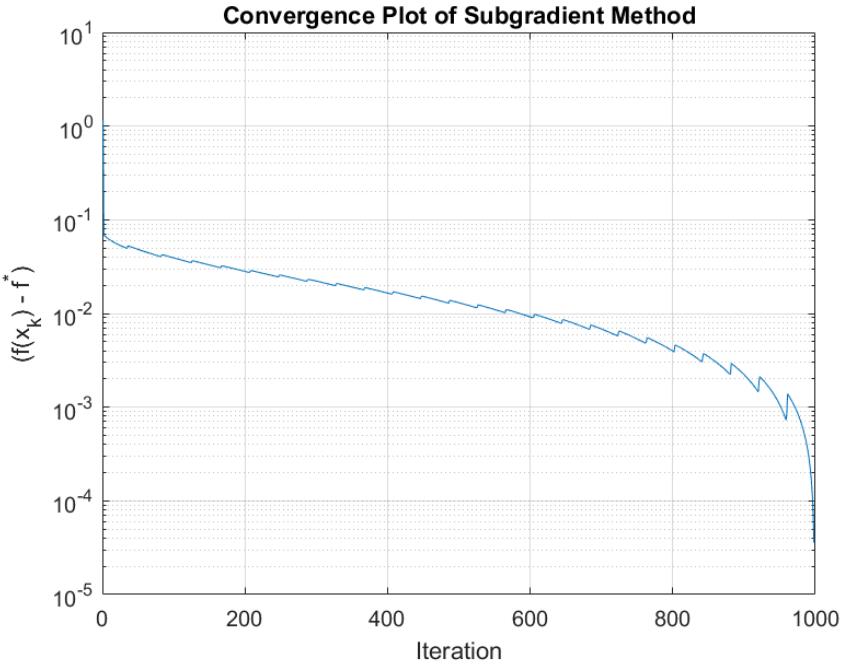

Figure 3: Convergence Plot of Subgradient Method

The convergence plot for the subgradient method shows a steady decrease in the objective value gap (f(x_k) − f*) over iterations. In the log-scale plot, we can clearly see a downward trend with small oscillations throughout, this is expected due to the non-smooth nature of the objective and the fact that we're not using exact gradients, the curve flattens as we approach higher iteration counts, which reflects the diminishing step size and the method's slow but steady progress. The shape of the curve suggests **sublinear** convergence, in particular, the $O(1/\sqrt{k})$ rate discussed earlier. While the convergence is certainly not fast, the method is stable and makes reliable progress toward the optimum, which is exactly what we would expect in this setting. Note: The plot shows the objective function values at each iteration, f(x_k), rather than the best value obtained so far. This is intentional because the theoretical convergence guarantees, (like the $O(1/\sqrt{k})$ rate) , are based on the iterates themselves, not on the minimum value observed along the way. By plotting f(x_k) – f*, we get a clearer picture

of how the method progresses, which aligns more closely with standard convergence analyses in subgradient methods.

Moreover, from a theoretical perspective, the subgradient method is expected to converge at a sublinear rate $O(1/\sqrt{k})$, when applied to convex problems with bounded subgradients. This theoretical result is based on a few core conditions being met: the objective function must be convex, subgradients should be bounded, and the step size must decrease over time in such a way that:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

These conditions are satisfied in the implementation, and the theoretical bound

$$f_{\text{best}}^{(k)} - f^* \leq \mathcal{O}\left(\frac{1}{\sqrt{k}}\right).$$

Can be assumed to hold here, given our conditions.

Empirically, the convergence plot of (f(x_k) − f*) versus iteration shows a decreasing trend with small oscillations and gradually stabilizing without any abrupt drops, this is consistent with the theoretical behavior of the subgradient method when using a diminishing step size. Throughout the run, the method maintained feasibility and did not encounter numerical instability, and importantly, the key theoretical assumptions behind the convergence rate are satisfied in our setup. The subgradients used in the method originate from the hinge loss and regularization terms, both of which depend on finite data, ensuring they remain bounded as required for the $O(1/\sqrt{k})$ guarantee, also, the convergence plot is based on raw iterates rather than averaged ones. To sum up, we can say that the empirical results align well with theoretical predictions, and no significant discrepancies were observed.

Additionally, from a numerical implementation perspective:

- **No trust region mechanism was used**:

  Trust region methods are essentially used in second-order optimization, like Newton or quasi-Newton algorithms, where you have some curvature information to build and trust a local model. In contrast, the subgradient method relies purely on first-order subgradients and can point in directions that don't follow the smooth descent we usually see in gradient-based methods. So, applying a trust region here wouldn't really make sense. It's not part of the method's logic, and there's nothing to "trust" in the way trust region methods are designed to do.

- **No step size of 1 or line search was used:**

  In methods like steepest descent or Newton's method (gradient-based methods), a step size of 1 can sometimes be taken if the direction is reliable, and line search is used to make sure the step leads to enough decrease in the function value (like the Wolfe conditions). But in subgradient methods, line search doesn't really apply because the function isn't smooth, so the usual assumptions about descent and curvature don't hold and that's why we use a diminishing step size instead.

- **Stopping Criterion:**

  While subgradient methods are generally not stopped using the subgradient norm (since subgradients can remain large near nonsmooth points), I included a basic safeguard to stop early if the norm of the subgradient became very small. Still, the main stopping condition was reaching the maximum number of iterations.

From a theoretical perspective, the subgradient method has a per-iteration complexity of $O(N)$, where N is the number of data points. This is because each iteration requires evaluating the hinge loss and computing the subgradient by summing over all margin-violating samples. In practice, this aligns with empirical observations: each iteration was completed efficiently, for 18 data points, and runtime scaled linearly with the number of iterations. That said, it's not just about how fast each step is, as we also need to consider how many steps are needed. The method has a sublinear convergence rate of $O(1/\sqrt{k})$, so to get a small gap in the objective value (say, ε), it can take around $O(1/\varepsilon^2)$ iterations, which means that even though each step is cheap, we might need a lot of them to get close to the optimum. Compared to second-order methods (like Newton's), which may take fewer steps but do more work per step, the subgradient method is the opposite (it trades high iteration count for low per-iteration cost) so overall, this balance between cheap steps and many iterations is what defines the complexity of the method.

As discussed previously, each iteration of the subgradient method has a very low computational cost, and the total runtime scales linearly with the number of iterations. This theoretical expectation was confirmed in practice: running the method for 1000 iterations on a dataset with 18 training points took roughly 0.0763 seconds (measured using MATLAB's tic/toc functions). This short runtime reflects the simplicity of the method where each step involves only basic vector operations, with no matrix factorizations, line search, or second-order computations, thus, the result is consistent with the subgradient method's known efficiency in terms of CPU usage.

Turning to memory usage, theoretically, its memory is efficient and has a footprint of $O(n)$, since only the current iterate x_k, the current subgradient g_k, and a few scalars (such as the step size and function value) need to be stored at each iteration. No Hessians or history matrices are maintained, which is in contrast to second-order methods like BFGS or Newton's, that require $O(n^2)$ storage for curvature approximations. In line with theoretical expectations, the method only stores the current iterate and subgradient at each step, making

its memory footprint minimal. However the only addition in the implementation was storing the loss history for plotting and analysis which, while useful for evaluation, is not required by the algorithm itself. In short, the subgradient method maintains a small memory footprint, making it a practical choice when storage is limited.

Total word count: 2850.

References

[1] C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

[2] J. Nocedal and S. Wright, Numerical Optimization, 2nd ed., Springer, 2006

[3] C. D. Manning, P. Raghavan, H. Schütze, Introduction to Information Retrieval, Cambridge University Press, 2008

[4] H. Xu, C. Caramanis, and S. Mannor, Robustness and Regularization of Support Vector Machines, Journal of Machine Learning Research, vol. 10, pp. 1485–1510, 2009

[5] S. Boyd and J. Park, Subgradient Methods, EE364b: Convex Optimization II, Stanford University, Spring 2014.

[6] M. M. Betcke, Numerical Optimisation – Nonsmooth Optimisation, Lecture 12, COMP0118, Department of Computer Science, University College London, Spring 2025.

[7] Bertsekas, D. P. (2010). Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization. MIT Lecture Slides.

[8] Convex Optimization (Boyd & Vandenberghe, 2004)

Appendix A

Subgradient Method for Solving the Primal Soft-Margin SVM (MATLAB)

Note: only the highlighted section corresponds to the implementation of the subgradient method, the rest is just plotting and printing statments.

```
function [model, loss_history, best_loss_history] =
subgrad_svm_method(data, labels, reg, max_steps)

% subgradient method for svm

% data          - training points (N x 2)

% labels        - class labels (+1 / -1)

% reg           - regularization parameter (C)

% max_steps     - number of iterations

% returns:

% model              - the final parameters [w; b]

% loss_history       - loss at each step

% best_loss_history - best loss seen so far at each step

[~, dim] = size(data); % gethow many features we have

model = zeros(dim + 1, 1);  % make a zero vector for[w; b]

loss_history = zeros(max_steps, 1);

best_loss_history = zeros(max_steps, 1);

best_loss = inf;

step_scale = 0.01;

tic; % measure CPU time

for step = 1:max_steps

    w = model(1:dim);

    b = model(end);

    % find which points are inside the margin

    margin_values = labels .* (data * w + b); % y(w*x+b) < 1

    hinge_points = find(margin_values < 1);

    % compute subgradient

    % w term is from the squared norm
```

```
    % and the hinge loss part is added using only the "hinge" points
    % sum of y_i * x_i for bad points
    grad_w = w - reg * sum(data(hinge_points, :) .* labels(hinge_points),
1)';
    grad_b = -reg * sum(labels(hinge_points));
    grad = [grad_w; grad_b];
    % update with diminishing step size
    step_size = step_scale / sqrt(step);
    model = model - step_size * grad;
    % % compute total loss: regularizer + hinge loss
    hinge = max(0, 1 - labels .* (data * model(1:dim) + model(end)));
    loss = 0.5 * norm(model(1:dim))^2 + reg * sum(hinge);
    loss_history(step) = loss;
    best_loss = min(best_loss, loss);
    best_loss_history(step) = best_loss;
    % stop if gradient is very small
    if norm(grad) < 1e-4 && step > 10
        fprintf('converged at step %d\n', step);
        loss_history = loss_history(1:step);
        best_loss_history = best_loss_history(1:step);
        break;
    end
end
cpu_time = toc;
fprintf('Total CPU time: %.4f seconds\n', cpu_time);
% check support vectors
w = model(1:dim);
b = model(end);
final_margins = labels .* (data * w + b);
tolerance = 1.5;
```

```
%find support vectors: points where y_i(w.T * x_i + b) = 1

support_indx = find(abs(final_margins - 1) < tolerance);

fprintf('support vectors at indices: %s\n', mat2str(support_indx));

fprintf('final weights: [%.4f %.4f], bias: %.4f\n', model(1), model(2),
model(3));

figure;

pos = labels == 1;

neg = labels == -1;

scatter(data(pos,1), data(pos,2), 50, 'r', 'o', 'filled'); hold on;

scatter(data(neg,1), data(neg,2), 50, 'b', 'x', 'LineWidth', 1.5); hold on;

grid on;

x_plot = linspace(min(data(:,1)) - 1, max(data(:,1)) + 1, 100);

y_plot = -(model(1) * x_plot + model(3)) / model(2); % decision boundary
equation of the SVM

plot(x_plot, y_plot, 'k-', 'LineWidth', 2);

margin1 = -(model(1)*x_plot + model(3) - 1) / model(2);

margin2 = -(model(1)*x_plot + model(3) + 1) / model(2);

plot(x_plot, margin1, 'k--');

plot(x_plot, margin2, 'k--');

scatter(data(support_indx,1), data(support_indx,2), 100, 'ko', 'LineWidth',
1.5);

txt = sprintf('w = [%.2f, %.2f], b = %.2f', model(1), model(2), model(3));

text(0, 2, txt, 'FontSize', 10, 'BackgroundColor', 'white', 'EdgeColor',
'black','FontSize', 8);

xlabel('Feature 1');

ylabel('Feature 2');

legend('+1', '-1', 'Decision Boundary', 'Margin +1', 'Margin -1', 'Support
Vectors','FontSize', 8);

title('SVM Decision Boundary with Subgradient Method', 'FontSize', 10);

hold off;

figure;

f_star = min(best_loss_history);  % best found value

semilogy(1:length(loss_history), loss_history - f_star);
```

```
xlabel('Iteration');

ylabel('(f(x_k) - f^*)');

title('Convergence Plot of Subgradient Method');

grid on;

end
```

Appendix B

Excerpt from Testing Script (MATLAB)

```
% calling test
X_pos = [3, 9;
         4, 8.5;
         5, 9.2;
         6.2, 7;
         4, 9.5;
         2.5, 10;
         3.5, 9.7;
         5.5, 8.8;
         4.8, 9.3];
X_neg = [7, 3;
         8, 2;
         9, 2.5;
         6.8, 6;
         8.5, 1.2;
         9.5, 1.7;
         7.8, 2.3;
         8.8, 1.5;
         9.2, 2.1];
X = [X_pos; X_neg];
y = [ones(size(X_pos,1),1); -ones(size(X_neg,1),1)]; % class +1 and -1 labels
C = 1;  % regularization parameter
max_iter = 1000;
[x, f_hist, f_best_hist] = subgrad_svm_method(X, y, C, max_iter);
```

```
>> subgrad_svm
Total CPU time: 0.0763 seconds
support vectors at indices: [4;10;13]
final weights: [-0.5847 0.6647], bias: -0.0274
```

***Figure B.1**: Console output from subgradient SVM confirming support vector detection and final model parameters.*