Numerical Optimisation

Semismooth Newton Support Vector Machines

Jacobo Ostos Bollmann

University College London

Throughout, denote by $\mathcal{M}_{m,n}(\mathbb{R})$ the set of all $m \times n$ matrices with entries in $\mathbb{R}$, if $m = n$ then we write $\mathcal{M}_n(\mathbb{R})$ instead. Let $\mathcal{S}_n$ denote the linear subspace of all symmetric matrices of $\mathcal{M}_n(\mathbb{R})$. Write $\mathcal{S}_n^+$ for the set of positive definite matrices of $\mathcal{M}_n(\mathbb{R})$, and $\overline{\mathcal{S}_n^+}$ for the closure of $\mathcal{S}_n^+$ in $\mathcal{M}_n(\mathbb{R})$, i.e., the set of all symmetric positive semidefinite matrices; use log for the natural logarithm, boldface uppercase, e.g., $\boldsymbol{A}$, for matrices, and lowercase, e.g., $\boldsymbol{a}$, for vectors, $\|\cdot\|_2$ for the l_2-norm and $\|\cdot\|_\infty$ for l_∞-norm, and $(\cdot)_+$ the positive part of a function.

1 SVM classification and the underlying optimisation problem

[a] The present work focuses on the binary classification problem discriminating breast cancer diagnostics. Formally, consider a dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y} : i \in \mathcal{I}\}$, where $\mathcal{X} \subseteq \mathbb{R}^d$ is the domain set of features we wish to label, $\mathcal{Y} \in \{-1, 1\}$ the binary label set of corresponding target classes, and $\mathcal{I} = \{0, \dots, n\}$ the index set of observations. Further, we assume there exists an unknown distribution, D_{XY}, defined on $\Omega = (\mathcal{X} \times \mathcal{Y})^n$, generating the labeled pairs $\{(\boldsymbol{x}_i, y_i)\}_{i \in \mathcal{I}}$. Then we are interested in learning a binary classifier, that is, a function $h : \mathcal{X} \to \mathcal{Y}$, strictly of $\mathcal{D}$. To do so, we will find the optimal hyperplane separating the two labels through a Support Vector Machine (SVM) algorithm, given by its normal vector. Particularly, we borrow the binary labeled breast cancer dataset in Wolberg (1992), where $d = 9$ are features in the input space, assigned $\{-1\}$ for benign (cancer) diagnosis, and $\{1\}$ malignant, with a total of $n = 699$ samples. In empirical settings, the dataset was split into training ($n_{train} \approx 0.8n$) and testing ($n_{test} \approx 0.2n$) subsets to asses the performance of the algorithm, where the feature set was normalised to unit Gaussian in both cases.

[b] Evidence regarding the non-separability of the dataset is strong (see Mangasarian and Wolberg (1990), Mangasarian et al. (1990), Bennett and Mangasarian (1992) among others), hence we require that the decision boundary be non-linear in formulating the SVM. A popular choice involves employing a Gaussian Radial Basis Function (RBF) kernel.

Definition 1.1 (Gaussian RBF Kernel Liu et al. (2011))**.** *Let $\boldsymbol{x}_i$, $\boldsymbol{x}_j \in \mathcal{X}$ be two samples of some input feature space, $\mathcal{X} \subseteq \mathbb{R}^d$, and $\gamma \in \mathbb{R}_+$ a hyperparameter. Then the function $K : \mathcal{X}^2 \to \mathbb{R}$ given by*

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2\right). \tag{1}$$

is called a Gaussian Radial Basis Function kernel.

Introducing an RBF kernel enables an implicit mapping of data into a higher (possibly infinite) dimensional space where linear separability may become available, yet, in doing so through dot products, pairwise point relationships are preserved (Schölkopf et al. (1997), Vert et al. (2004)). This is regarded as the *"kernel trick"*. Further, we consider a soft-margin formulation, meaning we allow misclassifications to enhance the model generalisation properties. The choice of a loss function is in this case the quadratically smoothed Hinge Loss, which has both linear and quadratic regions in its topology – this is further covered in more detail later.

Definition 1.2 (Squared Hinge Loss, Schölkopf et al. (2002))**.** *Let $\boldsymbol{x}_i \in \mathcal{X}$ denote a feature space input, and $y_i \in \mathcal{Y}$ its corresponding true label in some observation dataset, $\mathcal{D}$, and assume $h : \mathcal{X} \to \mathcal{Y}$ is a learned classifier approximating their distribution. Then the squared hinge loss function is defined as*

$$L(\boldsymbol{x}_i, y_i, h(\boldsymbol{x}_i)) = \max\left(0, 1 - y_i h(\boldsymbol{x}_i)\right)^2 = (1 - y_i h(\boldsymbol{x}_i))_+^2. \tag{2}$$

where $(\cdot)_+$ is the positive part of the function.

These considerations are gathered under the primal problem, in the form of the unconstrained Lagrangian as a feasible optimisation problem. Essentially, primal formulation will optimise a fixed subset of variables, so called working set (Abe (2009)).

[c] We now formalise this intuition of the primal, derived from the traditional least-squares SVM formulation. Starting from the linear case:

Definition 1.3 (Primal Soft-Margin Linear SVM Formulation with Squared Hinge Loss, Platt (1999), Vapnik (2000), Zhou et al. (2007))**.** *Consider a classification problem with given labeled dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y} : i \in \mathcal{I}\}$, of training inputs $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^d$ and corresponding labels $y_i \in \mathcal{Y} = \{-1, 1\}$. Noting matrices $\boldsymbol{A} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m]^\top$, and $\boldsymbol{D} = diag(\boldsymbol{y})$ where $\boldsymbol{y} = [y_1, \ldots, y_m]^\top$, the linear SVM classifying $\mathcal{D}$ through the optimal hyperplane solves the following primal quadratic programming with parameter $v \in \mathbb{R}_+$*

$$\begin{aligned} \min_{(\boldsymbol{w}, b, \boldsymbol{\xi}) \in \mathbb{R}^{(d+1+m)}} \; & \frac{1}{2}(\boldsymbol{w}^\top \boldsymbol{w} + b^2) + \frac{v}{2}\left[\|\boldsymbol{\xi}\|_2^2 - \sum_{i \in \mathcal{I}} (1 - y_i h(\boldsymbol{x}_i))_+^2\right] \\ \textit{subject to} \quad & \boldsymbol{D}(\boldsymbol{A}\boldsymbol{w} - \vec{\mathbf{1}}b) \geq \vec{\mathbf{1}} - \boldsymbol{\xi}, \end{aligned} \tag{3}$$

where $\vec{\mathbf{1}}$ is a ones vector, $\boldsymbol{\xi} \in \mathbb{R}^k$ the slack variables, $\boldsymbol{w} \in \mathbb{R}^d$ the weight normal vector defining the hyperplane, and b the bias term. Further, noting $\boldsymbol{H} = \boldsymbol{D}[\boldsymbol{A} \quad \vec{\mathbf{1}}]$, and $\boldsymbol{s} = [\boldsymbol{w} \quad b]^\top$, the

unconstrained minimisation problem defining the Lagrangian of (3) *may be expressed as*

$$\min_{\boldsymbol{u}\in\mathbb{R}^{d+1}} \mathcal{L}(\boldsymbol{s}) := \frac{1}{2}(\boldsymbol{s}^\top \boldsymbol{s}) + \frac{v}{2}\left[(\vec{\mathbf{1}} - \boldsymbol{H}\boldsymbol{s})^\top (\vec{\mathbf{1}} - \boldsymbol{H}\boldsymbol{s})_+\right]. \tag{4}$$

We follow with the non-linear, RBF kernel formulation, and discuss the differences with the standard primal.

Definition 1.4 (Primal Soft-Margin Non-Linear SVM Formulation with RBF Kernel and Squared Hinge Loss, Lin and Saigal (2000) Fine and Scheinberg (2001), Ferris and Munson (2004), Zhou et al. (2007))**.** *Following the classification problem setup in* **Definition 1.3***, the non-linear SVM classifying $\mathcal{D}$ through the optimal hyperplane solves the following* (*constrained*) *primal quadratic programming*

$$\begin{aligned} \min_{(\boldsymbol{w},b,\boldsymbol{\xi})\in\mathbb{R}^{(d+1+m)}} \ & \frac{1}{2}(\boldsymbol{w}^\top \boldsymbol{w} + b^2) + \frac{v}{2}\left[\|\boldsymbol{\xi}\|_2^2 - \sum_{i\in\mathcal{I}}(1 - y_i h(\boldsymbol{x}_i))_+^2\right] \\ \textit{subject to} \quad & \boldsymbol{D}(K(\boldsymbol{A},\boldsymbol{A}^\top)\boldsymbol{w} - \vec{\mathbf{1}}b) \geq \vec{\mathbf{1}} - \boldsymbol{\xi}, \end{aligned} \tag{5}$$

where $K(\cdot,\cdot)$ is the RBF kernel as in **Definition 1.1***. Then consider the incomplete Cholesky factorisation of the kernel matrix $\boldsymbol{G}\boldsymbol{G}^\top \approx K(\boldsymbol{A},\boldsymbol{A}^\top)$, and noting $\hat{\boldsymbol{H}} = \boldsymbol{D}[\boldsymbol{G} \quad \vec{\mathbf{1}}]$, and $\boldsymbol{s} = [\boldsymbol{w} \quad b]^\top$, the unconstrained minimisation problem defining the Lagrangian of* (5) *may be expressed as*

$$\min_{\boldsymbol{s}\in\mathbb{R}^{d+1}} \hat{\mathcal{L}}(\boldsymbol{s}) := \frac{1}{2}(\boldsymbol{s}^\top \boldsymbol{s}) + \frac{v}{2}\left[(\vec{\mathbf{1}} - \hat{\boldsymbol{H}}\boldsymbol{s})^\top (\vec{\mathbf{1}} - \hat{\boldsymbol{H}}\boldsymbol{s})_+\right]. \tag{6}$$

Remark 1.1. *Formulations given in* **Definitions 1.3** & **1.4***, particularly in terms of their Lagrangians, only are discerned by $\boldsymbol{H}, \hat{\boldsymbol{H}}$, where $\hat{\boldsymbol{H}}$ will be defined on a higher-dimensional space due to the kernel uplift* (Zhou et al. (2007))*.*

Contrary to standard formulations of the primal SVM (for instance, see Vapnik (2000), Lee and Mangasarian (2001)) with plain (i.e., non-squared) hinge loss, we employ the squared 2 norm of both bias, b, and slack variables, $\boldsymbol{\xi}$. This eliminates non-negativity constraints, and enhances the convexity and smoothness properties of the objective function. Further, it penalises the bias b, and we can optimise for the hyperplane location as well. By introducing a kernel, we allow non-linearity of the decision boundary; Finally, two aspects are crucial to optimisation, the problem is proposed as a (once) differentiable function, and eliminating constrains allows employing Newton's method. We cover function topology below.

[d] Immediately, we notice the SVM formulation above (6), is cast as a quadratic programming, being an unconstrained and piecewise quadratic minimisation problem finding optimal

$\boldsymbol{s}$. We are then interested in showing its strict convexity.

Definition 1.5 (Convex Function, Nocedal and Wright (2006))**.** *A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is convex if $\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$, $\forall \theta \in (0, 1)$, it holds that:*

$$f(\theta \boldsymbol{x} + (1 - \theta)\boldsymbol{y}) \leq \theta f(\boldsymbol{x}) + (1 - \theta) f(\boldsymbol{y}). \tag{7}$$

Further, if the inequality is strict, with $\boldsymbol{x} \neq \boldsymbol{y}$ and such that

$$f(\theta \boldsymbol{x} + (1 - \theta)\boldsymbol{y}) < \theta f(\boldsymbol{x}) + (1 - \theta) f(\boldsymbol{y}), \tag{8}$$

we say f strictly convex.

Then, consider the general squared Euclidean norm ($\|\cdot\|_2^2$) corresponding to the first term in (6). Let $\theta \in (0, 1)$ and assume $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{X}$ s.t. $i \neq j \in \mathcal{I}$, are two arbitrary feature space inputs. We have that, under convex combination,

$$\begin{aligned}
& \theta\|\boldsymbol{x}_i\|_2^2 + (1 - \theta)\|\boldsymbol{x}_j\|_2^2 - \|\theta \boldsymbol{x}_i + (1 - \theta)\boldsymbol{x}_j\|_2^2 \\
= \; & \theta\|\boldsymbol{x}_i\|_2^2 + (1 - \theta)\|\boldsymbol{x}_j\|_2^2 - \theta^2\|\boldsymbol{x}_i\|_2^2 - (1 - \theta)^2\|\boldsymbol{x}_j\|_2^2 - 2\theta(1 - \theta)\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle \\
= \; & \theta(1 - \theta)(\|\boldsymbol{x}_i\|_2^2 + \|\boldsymbol{x}_j\|_2^2 - 2\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle) \\
= \; & \theta(1 - \theta)\|\boldsymbol{x}_i - \boldsymbol{x}_i\|_2^2 > 0,
\end{aligned}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product, and hence $\|\cdot\|_2^2$ strictly convex. Now, consider the squared hinge loss function in **Definition 1.2**, corresponding to the remaining part of (6). Through its piecewise definition, standard (i.e., non-squared) hinge loss is a convex function since every linear function is convex. Then the quadratically smoothed hinge loss as well since the square of any such convex function is again convex, though remains not strictly (due to the maximum function) (Beckenbach (1948)). Then, in combining both squared Euclidean norm and (squared) hinge loss, we know the sum of a convex function and strictly convex function is strictly convex, since strict inequality holds in at least one case, thus so is (6). Another important result states that for any strictly convex function, every local minimiser is also a global minimiser, and for our (once) differentiable objective function, $\hat{\mathcal{L}}$, any stationary point will be a global minimiser. Finally, relating to smoothness, squared hinge loss is not twice differentiable since it is not differentiable at 0, posing a challenge in optimisation; we later show its strong semismoothness. We propose an unconstrained piecewise quadratic programming, with strictly convex objective function that is once but not twice differentiable.

2 Optimisation method and convergence theory

[a] The employed method is the Conjugate Gradient Semismooth Newton SVM algorithm (SNSVM)(Yin and Li (2019)). To understand this choice we introduce semismoothness of functions.

Definition 2.1 (Semismooth function, Clarke (1983), Ferris and Munson (2004), Zhou et al. (2007))**.** *Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a locally Lipschitzian but not necessarily continuously differentiable function, and $\partial f(\boldsymbol{x})$ the Clarke generalised Jacobian of f at $\boldsymbol{x} \in \mathbb{R}^n$. Then, for $\boldsymbol{V} \in \partial f(\boldsymbol{x} + \boldsymbol{d})$, $\boldsymbol{x}, \boldsymbol{d} \in \mathbb{R}^n$, we say f is semismooth at $\boldsymbol{x}$ if $\boldsymbol{V}\boldsymbol{d} - \boldsymbol{J}(\boldsymbol{x}, \boldsymbol{d}) = \varphi(\|\boldsymbol{d}\|)$, when $\|\boldsymbol{d}\| \to 0$, and where $\boldsymbol{J}(\cdot, \cdot)$ is the directional derivative of f at $\boldsymbol{x}$ in the $\boldsymbol{d}$ direction. $\varphi(\|\cdot\|)$ denotes a vector-valued function $\varphi : \mathbb{R}^n \to \mathbb{R}^n$ satisfying $\lim_{d\to 0} \varphi(d)/\|d\| = 0$. Further, f strongly semismooth at $\boldsymbol{x}$ if $\boldsymbol{V}\boldsymbol{d} - \boldsymbol{J}(\boldsymbol{x}, \boldsymbol{d}) = \phi(\|\boldsymbol{d}\|)$, where this time $\phi : \mathbb{R}^n \to \mathbb{R}^n$ satisfies $\|\phi(\boldsymbol{d})\| \leq M\|\boldsymbol{d}\|^2$, $\boldsymbol{d} \in \mathbb{R}^n$, $M \in \mathbb{R}_+$, and $\|\boldsymbol{d}\| \leq \delta$, $\delta \in \mathbb{R}_+$.*

Lemma 2.1. *The function $\psi(a) = a_+ = \max(a, 0)$ is strongly semismooth* (Ferris and Munson (2004), Zhou et al. (2007))*.*

Now, the gradient of our objective formulation in (6), given by

$$\nabla\hat{\mathcal{L}}(\boldsymbol{s}) := \boldsymbol{s} - v\hat{\boldsymbol{H}}^\top(\vec{\boldsymbol{1}} - \hat{\boldsymbol{H}}\boldsymbol{s})_+ \tag{9}$$

yields a system of equations $\boldsymbol{s} - v\hat{\boldsymbol{H}}^\top(\vec{\boldsymbol{1}} - \hat{\boldsymbol{H}}\boldsymbol{s})_+ = 0$ to find its unique minima (recalling convexity), which in turn is a system of linear non-differentiable equations defined piecewise. From **Lemma 2.1**, and by **Definition 2.1** we can immediately tell $\nabla\hat{\mathcal{L}}$ strongly semismooth. The solution of these equations is indeed that of the programming in (6), and will require a semismooth method, hence our choice. Further Newton's method opens the possibility for fast (quadratic) convergence – conditions are covered in sections **[c, d]** – where inability to compute the Hessian, $\nabla^2\hat{\mathcal{L}}$, is handled through subdifferentials since strong semismoothness is satisfied (Clarke (1983)). Additionally, strict convexity and uniqueness of global minima make starting point guesses trivial. Computationally, we outline this algorithm only needs to minimise a low-dimensional differentiable unconstrained convex piecewise qudratic objective, where speedy convergence can be expected, with lower training times and prowess for larger scale optimisation problems (low memory requirements). So is done by casting the low dimensional equivalent of Mangasarian's perturbation quadratic program.

[**b**] Briefly, we outline the logic in **Algorithm 1**.

Proposition 2.1 (Semismooth Newton SVM (Yin and Li (2019))). *Let $\hat{\boldsymbol{H}} \in \mathbb{R}^{l+1}$ denote a lifted input matrix from a dataset we wish to classify, and it is given according to 1.4 through a Gaussian RBF kernel. Then the algorithm is initialised with initial hyperplane normal vector and bias $\boldsymbol{s}_0 = [\boldsymbol{w}_0 \quad b_0]^\top \in \mathbb{R}^{d+1}$, parameters $v, \epsilon, \beta \in \mathbb{R}_+$, $p > 2$, and $\sigma \in (0, \frac{1}{2})$, and at each iteration with concurrent generalised Clarke Jacobian of the objective gradient (semismooth Hessian surrogate) $\boldsymbol{B}_k \in \partial\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)$, $k \in \mathbb{N}$, the new descent direction $\boldsymbol{d}_k \in \mathbb{R}^{d+1}$ is obtained by solving the Newton system*

$$\boldsymbol{B}_k \boldsymbol{d} = -\boldsymbol{g}_k \tag{10}$$

where $\boldsymbol{g}_k = \nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)$ is the corresponding value of the objective gradient at $\boldsymbol{s}_k$. If (10) is unsolvable or violates the descent condition imposed by $\|\boldsymbol{B}_k\boldsymbol{d}_k + \nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)\| \leq \mu_k\|\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)\|$, with $\mu_k = \min(\eta_1, \eta_2\|\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)\|)$, then the descent direction is rejected and a new generalised gradient is computed to solve the (Newton) system. For such descent direction, the optimal step size is given through Armijo backtracking algorithm (see (12) in **Algorithm 1**), *and the hyperplane normal vector and bias, $\boldsymbol{s}_{k+1} = \boldsymbol{s}_k + \alpha_k\boldsymbol{d}_k$, and objective gradient $\boldsymbol{g}_{k+1} = \nabla\hat{\mathcal{L}}(\boldsymbol{s}_{k+1})$ are updated. Whenever the optimal normal vector and bias, $\boldsymbol{s}^*$, is achieved, and potential progress measured by the gradient norm falls below some tolerance, $\|g_k\| < \epsilon$, the process is terminated.*

Most importantly, we the following characterisation of $\boldsymbol{B}_k$.

Theorem 2.1 (Zhou et al. (2007)). *For $\nabla\hat{\mathcal{L}}(\boldsymbol{s})$ as in (9), it holds that the generalised Clarke Jacobian is given by $\partial\nabla\hat{\mathcal{L}}(\boldsymbol{s}) \subset \{\boldsymbol{I} + v\hat{\boldsymbol{H}}^\top\boldsymbol{D}\hat{\boldsymbol{H}}\}$, where $\boldsymbol{D} = diag[u_1, \ldots, u_m]$ with each $u_i = \partial\Phi(1 - \hat{\boldsymbol{H}}_i\boldsymbol{s})$, $i = 1, \ldots, m$, and every $\boldsymbol{B} \in \partial\nabla\hat{\mathcal{L}}(\boldsymbol{s}) \in \mathcal{S}_n^+$ is positive definite.*

For the solution of the linear Newton systems we choose the classical Hestenes-Stiefel Conjugate Gradient method. As a CG method, it has the characteristic of combining weighted information about the current and previous iterate's gradient to find the new descent directions, and afterwards the minimiser is found along it through a line search method, in this case Backtracking Armijo line search. The implementation is clear in **Appendix A.4**, and otherwise extensively covered in Hestenes and Stiefel (1952), and Nocedal and Wright (2006) pp.121-133[1].

[1] As a classic method, prominent in the literature and widely available from a number of reputable sources (of which we provide the two above and our own implementation in the **Appendix**), we avoid directly covering its mathematical formulation due to space requirements.

Algorithm 1 Semismooth Newton SVM (Yin and Li (2019))

Input: Data $\hat{\boldsymbol{H}} \in \mathbb{R}^{l+1}$
Parameters: Initial point $\boldsymbol{s}_0 \in \mathbb{R}^{l+1}$, and define $v, \epsilon, \beta \in \mathbb{R}_+$, $p > 2$, and $\sigma \in (0, \frac{1}{2})$
Result: Optimal set of weights and bias for separating non-linear hyperplane, $\boldsymbol{s}^*$

$\boldsymbol{g}_0 \leftarrow \nabla\hat{\mathcal{L}}(\boldsymbol{s}_0)$ gradient at $\boldsymbol{s}_0$
$k \leftarrow 0$
repeat
  if $\|g_k\| \leq \epsilon$ **then**
    terminate
  end if
  while $\boldsymbol{B}_k\boldsymbol{d} = -\boldsymbol{g}_k$ unsolvable or **not** satisfies the descent condition

$$\|\boldsymbol{B}_k\boldsymbol{d}_k + \nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)\| \leq \mu_k\|\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)\|, \quad \text{with } \mu_k = \min(\eta_1, \eta_2\|\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)\|) \tag{11}$$

  do
    Let $\boldsymbol{B}_k \in \partial\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)$, and calculate $\boldsymbol{d}_k$ by solving the Newton system $\boldsymbol{B}_k\boldsymbol{d} = -\boldsymbol{g}_k$ through CG.
  end while
  Line search: (Armijo backtracking) choose $\alpha_k = 2^{-i_k}$, i_k is the smallest integer such that

$$\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k + 2^{-i_k}\boldsymbol{d}_k) \leq \nabla\hat{\mathcal{L}}(\boldsymbol{s}_k) + \sigma 2^{-i_k}\boldsymbol{g}_k^\top\boldsymbol{d}_k \tag{12}$$

  $\boldsymbol{s}_{k+1} \leftarrow \boldsymbol{s}_k + \alpha_k\boldsymbol{d}_k$
  $\boldsymbol{g}_{k+1} \leftarrow \nabla\hat{\mathcal{L}}(\boldsymbol{s}_{k+1})$
  $k \leftarrow k + 1$
until $\|g_k\| < \epsilon$

[c, d] Briefly, in terms of type of convergence we propose the following theorem;

Theorem 2.2 (Semismooth Newton Global Convergence, Hintermüller (2010) pp.18-19)**.** *Suppose that f is locally Lipschitz continuous semismooth on U_0, the closure of $\mathcal{B}(x_0, r)$ (ball). Also, suppose that for any $G(x) \in \partial f(x)$, $x \in U_0$, $G(x)$ is nonsingular, and, with $y \in U_0$,*

$$\|G(x)^{-1}\| \leq \beta, \quad \|G(x) - f'(x; y-x)\| \leq L\|y-x\|,$$
$$\|f(y) - f(x) - f'(x; y-x)\| \leq \eta\|y-x\|$$

where $\alpha = \beta(L+\eta) < 1$ and $\beta\|f(x_0)\| \leq r(1-\alpha)$. Then the iterates $\{x_k\}$ of the semismooth Newton algorithm remain in U_0 and converge to the unique solution x^* of $f(x) = 0$ in U_0. Further, it holds that

$$\|x_k - x^*\| \leq \frac{\alpha}{1-\alpha}\|x_k - x_{k-1}\|, \text{ for } k = 1, 2, \ldots$$

Theorem 2.2 is an extension on the classical Newton-Kantorovich Theorem. Particularly, now in order that $\alpha < 1$, smallness of L, η is required; smallness of L is induced where the diameter of ∂f is small for $\forall x \in U_0$. For our $\nabla\hat{\mathcal{L}}$, local lipschitzness is trivially met by strong semismoothness proved above (see result in Hintermüller (2010)), **Thm. 2.9**, pp.16-17). Then for non-singularity of $\boldsymbol{B}_k \in \partial\nabla\hat{\mathcal{L}}(\boldsymbol{s})$, it is enough to show **Theorem 2.1** holds.

Proof. Consider a vector-valued function $V(\boldsymbol{x}) = (\vec{\mathbf{1}} - \boldsymbol{H}\boldsymbol{x})_+$, satisfying

$$V_i(\boldsymbol{x}) = (\vec{\mathbf{1}} - \boldsymbol{H}_i\boldsymbol{x})_+, \quad i = \{1, 2, \ldots, m\};$$

and assume $\partial_j V_i(\boldsymbol{x})$ denotes the j^{th} element of our functions generalised gradient. We have

$$\partial_i V_i(\boldsymbol{x}) \subset -\boldsymbol{H}_{i,j}\partial\Phi(\vec{\mathbf{1}} - \boldsymbol{H}_i\boldsymbol{x})_+) = \boldsymbol{H}_{i,j}\boldsymbol{u}_i, \quad j = \{1, 2, \ldots, l+1\}$$

by the Chain Rule (Clarke (1983), **Thm. 2.3.9**, pp.42-43). Thus, $\partial V_i(\boldsymbol{x}) \subset -\boldsymbol{H}_i^\top \boldsymbol{u}_i$, and the Clarke generalised Jacobian of V satisfies

$$\partial V(\boldsymbol{x}) \subset -[\boldsymbol{s_1}\boldsymbol{H_1} \quad \boldsymbol{s_2}\boldsymbol{H_2} \quad \ldots \quad \boldsymbol{s_m}\boldsymbol{H_m}]^\top = -S(\boldsymbol{x})\boldsymbol{H}.$$

Then we have shown that $\partial\nabla\hat{\mathcal{L}}(\boldsymbol{s}) \subset \{\boldsymbol{I} + v\hat{\boldsymbol{H}}^\top \boldsymbol{D}\hat{\boldsymbol{H}}\}$, where $\boldsymbol{D} = diag[u_1, \ldots, u_m]$ with each $u_i = \partial\Phi(1 - \hat{\boldsymbol{H}}_i\boldsymbol{s})$, $i = 1, \ldots, m$, and every $\boldsymbol{B} \in \partial\nabla\hat{\mathcal{L}}(\boldsymbol{s})$ is positive definite (i.e., $\boldsymbol{B} \in \mathcal{S}_n^+$) (Zhou et al. (2007)). □

The properties proved so far as well serve to narrow the theoretical convergence rate which may be expected for the CG SNSVM. We give the following theorem:

Theorem 2.3 (Semismooth Newton Convergence Rate, Hintermüller (2010))**.** *Suppose that $f(x^*) = 0$ and that $\forall G(x) \in \partial f(x)$ are non-singular. Then the generalised Newton method (10) is Q-superlinearly convergent in a neighbourhood of x^*, i.e., $\mathcal{N}(x^*)$, if f semismooth at x^*, and quadratically convergent if f strongly semismooth at x^*.*

Theorems 2.2, **2.3** generalise the convergence results of classical Newton method where differentiability of f need not be assumed. Hence, for our problem we will expect guranteed global convergence with *Q-quadratic* rate, given local lipschitzness and strong semismoothness of our $\nabla\hat{\mathcal{L}}$, and the proven non-singularity of all $\boldsymbol{B}_k \in \partial\nabla\hat{\mathcal{L}}$. However, anticipating any empirical results, we observed that unless unit step size was achieved, only *Q-superlinear* convergence was achieved.

3 Solution and discussion of the results

[a] We effectively employed CG SNSVM (**Algo. 1**) to discriminate breast cancer diagnosis based on the dataset, $\mathcal{D}$, characterised in **Section 1**. The relevant parameters employed are tabulated below.

Parameter	Value
σ	$1e^{-4}$
$\boldsymbol{s}_0$	$\vec{\mathbf{1}}$
δ	$1e^{-8}$
η_1	$5 \times 1e^{-2}$
η_2	$3 \times 1e^{-1}$
α_0	1
C	$(1/dim(\boldsymbol{s})) \times 1e^2$
Semismooth Newton Maximum Iterations	$1e^2$
CG Phase Maximum Iterations	$1e^3$

Table 1: Parameter Choice for Conjugate Gradient Semismooth Support Vector Machine, **Algorithm 1**.

Briefly, σ (commonly c_1) is the *Armijo Backtracking* sufficient decrease condition parameter; $\boldsymbol{s}_0 = [\boldsymbol{w}_0 \quad b_0]^\top$ the starting point (hyperplane weights and location parameter/bias), δ the gradient norm ($\|\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)\|$) tolerance to monitor the evolution of dependent variables, $\hat{\mathcal{L}}(\boldsymbol{s}_k)$, with respect to independent ones $\boldsymbol{s}_k$; η_1, η_2 are parameters checking whether concurrent (k^{th} iterate) search directions $\boldsymbol{d}_k$ resulting from the CG solutions of newton systems $\boldsymbol{B}_k\boldsymbol{d}_k = -\nabla\hat{\mathcal{L}}(\boldsymbol{s}_k)$ are indeed descent directions minimising the objective; α_0 is the step length (search direction scaling as usual) for line search block (despite in **Appendix A.4** $\alpha_0 = 0.5$ this is because the first exponent prior to Armijo backtracking update $i = 0 \Rightarrow \alpha_0^0 = (0.5)^0 = 1$); and then we set a maximum number of iterations as stopping criterion for both CG and Semismooth (generalised) Newton algorithms. We keep the discussion in this section succint, since performance overlaps with a critical assessment of results and those are covered in **[e]**.

[b, c] Let us introduce convergence sequences. To investigate convergence, we build sequences $r_k = \{||\boldsymbol{s}_k - \boldsymbol{s}^*||\} \in \mathbb{R}$ indexed by $k = 1, ..., N$ for N iterations, denoting the k^{th} normed error of sequences converging to the optimal hyperplane weights and bias, $\boldsymbol{s}^*$. Additionally, we consider *Q-convergence* sequences similarly given for $k = 1, \ldots, N$ iterations by $\frac{\|\boldsymbol{s}_{k+1} - \boldsymbol{s}^*\|}{\|\boldsymbol{s}_k - \boldsymbol{s}^*\|^2}$ of order $q = 2$.

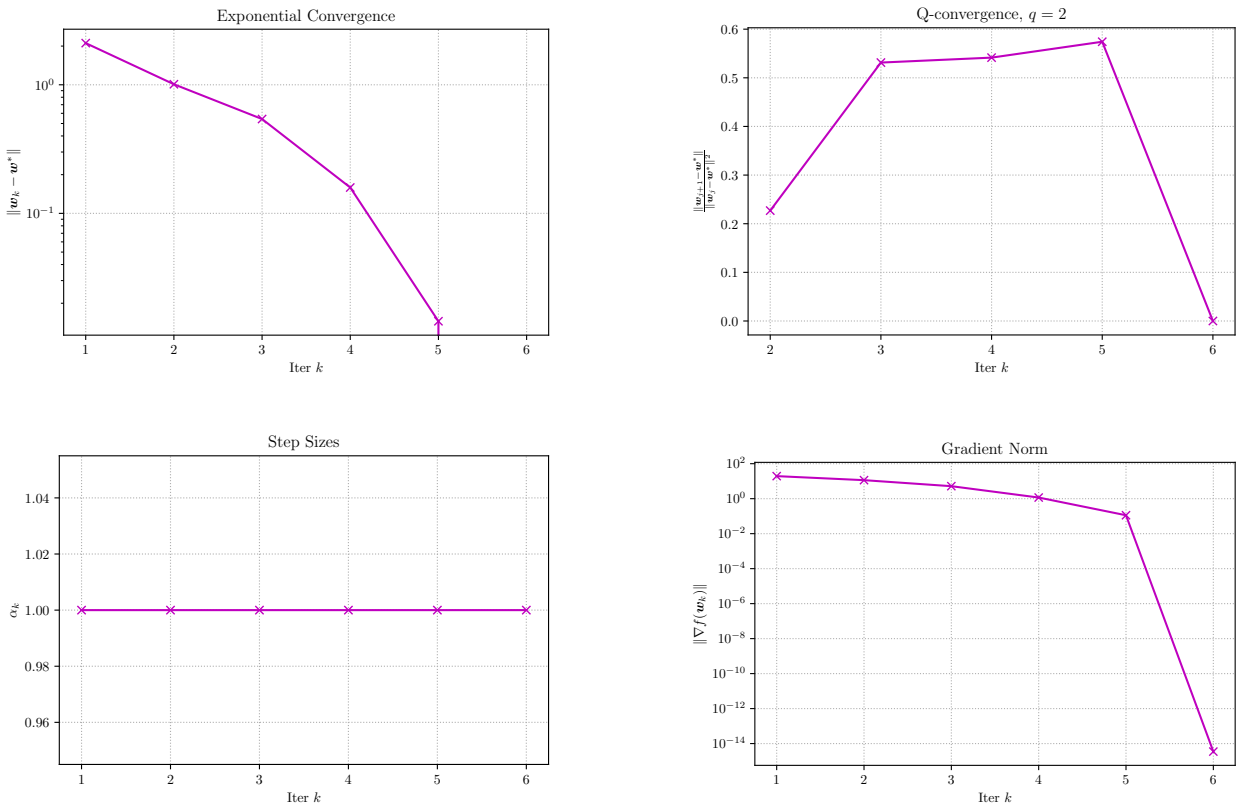

Figure 1: Convergence Relevant Plots: Exponential Convergence, Q-Convergence, Step Size, Gradient Norm over iterations

Starting from rate of convergence. Based on the exponential and Q-convergence plots in **Figure 1**, we observe CG SNSVM algorithm is *Q-quadratically* convergent, that is, in sequences $Cq^{2^k} \geq \|\boldsymbol{s}_{k+1} - \boldsymbol{s}^*\|$, in agreement with the stated theoretical prediction and empirical results in Yin and Li (2019). Now, such speedy convergence rate is nuanced in the case of Newton-type methods. Firstly, Q-quadratic rate is evidenced in that the Exponential convergence semi-log plot is faster than linear and superlinear. Q-convergence plot corroborates this result, where for $1 < q = 2$ the sequence remains bounded above by $\approx 0.6 (< 1)$ (since we are taking the empirical minima and not the true unknown minima, the sequence would converge regardless to zero, but we want to see its boundedness and stability for the order q, to satisfy such Q-convergence rate). Further, we must consider the sequence of step sizes $\{\alpha_k\}_{k \leq N}$. For generalised Newton directions, once we are in a sufficiently small neighbourhood of the minimiser, and if nonsingularity of our $\boldsymbol{B}_k$ is satisfied, quadratic convergence is recovered for unit step size. Now, we have already shown above that $\boldsymbol{B}_k \in \hat{\mathcal{L}}(\boldsymbol{s}_k)$ is always positive definite ($\boldsymbol{B}_k \in \mathcal{S}^+$), hence, non-singular. Thus it is sufficient to check that the unit step size is achieved, trivially satisfying the result. In this case, the unit step size is

maintained for $\forall k$, meaning the entire minimisation takes place with quadratic convergence. This is closely related to our starting point. The intuition here is that we will have unit (scaled) step sizes where the descent direction is aligned with the minimiser and/or the confidence in the search direction is significant. Recall, in **Section 1** that we have previously normalised dataset features to unit Gaussian (i.e., taking z-scores) and consider our initial guess, $\boldsymbol{s}_0 = [\boldsymbol{w}_0 \quad b_0]^\top = \vec{\mathbf{1}}$. Then we know we will be close to our minimiser and, due to convexity of the objective, any (accepted) descent direction shall provide sufficient confidence, and we can explain such behaviour. Summarising, our initial guess is always good given normalisation, while problem formulation and Clarke generalised Jacobian properties ensure the remaining convergence conditions are met, and quadratic convergence is recovered with unit step sizes. Finally, for convergence type, and again through strict convexity of $\hat{\boldsymbol{\mathcal{L}}}$, we know any stationary point will be a local and thus (through uniqueness) global minima. It is then enough to check that the gradient norm $\|\nabla\hat{\boldsymbol{\mathcal{L}}}\| \to 0$, converges. We see in **Figure 1** this is the case, i.e., such that for $\boldsymbol{s}_N \approx \boldsymbol{s}^*$ (true minima) we have $\|\nabla\hat{\boldsymbol{\mathcal{L}}}(\boldsymbol{s}_N)\| \approx 0 \Leftrightarrow \nabla\hat{\boldsymbol{\mathcal{L}}}(\boldsymbol{s}_N) \approx \vec{\mathbf{0}}$ and $\boldsymbol{s}_N$ approximates the stationary point.

[d] In semismooth Newton methods, the main concerns in terms of complexity are the elevated cost of computing the generalised Jacobian, and solution of the linear system for the Newton direction updates. Following the implementaion in Yin and Li (2019), we leverage matrix sparsity and direct computations in order to reduce the overall cost. Particularly, we avoid storing $\boldsymbol{B}_k$ and then computing the new update, which has a computational complexity $\mathcal{O}(n(l+1)^2)$ (n samples, $l = dim(\boldsymbol{s})$); and instead directly compute the updates $\boldsymbol{B}_k\Delta\boldsymbol{s}$ such that complexity is reduced to $\mathcal{O}(n|\boldsymbol{I}_k|)$, where $|\boldsymbol{I}_k| \ll (l+1)^2$ (see **A.5** in the **Appendix**; particularly for our Object Oriented Programming (OOP) approach it will correspond to the $V()$ methods for the function pointer classes: Linear, Polynomial kernel, RBF Gaussian kernel). The interested reader is encouraged to refer to Yin and Li (2019) for the full working of the result; but generally, the method is considered less complex and with a more parsimonious usage of computational resources than primitive implementations. As far as CPU time, the method took, on average on a MacBook Pro 16in. M1 (CPU) with 16gb RAM, 2.7 seconds.

[e] In critically discussing the results, we have successfully found a hyperplane with non-linear decision boundary separating the benign and malignant cancer diagnosis. In quantifying the goodness of these results, we outline the following metrics which have been tabulated below in **Table 2**. We refer to accuracy as the fraction of true results (true positives, true negatives) out of the total test samples; recall or sensitivity as the fraction of true positives out of

total positives; precision as the fraction of true positives out of all predicted positives; and, F1-score as the harmonic mean relating recall and precision. Finally, we also consider the area under the curve (AUC) for Receiver Operating Curve (ROC) measuring the models' discriminative performance.

Target Class	**Precision**	**Recall**	**F1-Score**
-1	1.00	0.93	0.96
1	0.81	1.00	0.89

Table 2: Classification Report for Target Classes.

In the macro average, we find an accuracy of $94,3\%$, with an AUC for ROC of 0.962. The discrimatory perfomance of the model is acceptable, nonetheless the purpose of this exercise was rather to explore the mathematical formulation and convergence behaviour of the implemented optimisation method. In this regard, we have also effectively explored the logic and convergence of our choice of CG SNSVM.

References

Abe, S. (2009). Is primal better than dual. pages 854–863.

Beckenbach, E. F. (1948). Convex functions. *Bulletin of the American Mathematical Society*, 54(5):439–460.

Bennett, K. P. and Mangasarian, O. L. (1992). Robust linear programming discrimination of two linearly inseparable sets. *Optimization Methods and Software*, 1:23–34.

Clarke, F. H. (1983). *Optimization and Nonsmooth Analysis*. John Wiley & Sons, New York.

Ferris, M. C. and Munson, T. S. (2004). Semismooth support vector machines. *Mathematical Programming Series B*, 101:185–204.

Fine, S. and Scheinberg, K. (2001). Efficient svm training using low-rank kernel representations. *Journal of Machine Learning Research*, 2:243–264.

Hestenes, M. R. and Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436.

Hintermüller, M. (2010). Semismooth newton methods and applications. *Oberwolfach-Seminar on "Mathematics of PDE-Constrained Optimization" at Mathematisches Forschungsinstitut in Oberwolfach*. Department of Mathematics, Humboldt-University of Berlin, hint@math.hu-berlin.de.

Lee, Y.-J. and Mangasarian, O. L. (2001). Ssvm: A smooth support vector machine. *Computational Optimization and Applications*, 20:5–22.

Lin, C.-J. and Saigal, R. (2000). An incomplete cholesky factorization for dense matrices. *BIT Numerical Mathematics*, 40:536–558.

Liu, Q., Chen, C., Zhang, Y., and et al. (2011). Feature selection for support vector machines with rbf kernel. *Artificial Intelligence Review*, 36:99–115.

Mangasarian, O. L., Setiono, R., and Wolberg, W. H. (1990). Pattern recognition via linear programming: Theory and application to medical diagnosis. In Coleman, T. F. and Li, Y., editors, *Large-Scale Numerical Optimization*, pages 22–30. SIAM Publications, Philadelphia.

Mangasarian, O. L. and Wolberg, W. H. (1990). Cancer diagnosis via linear programming. *SIAM News*, 23(5):1 & 18.

Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer, New York.

Platt, J. C. (1999). Fast training of support vector machines using sequential minimal optimization. In Schölkopf, B., editor, *Advances in Kernel Methods-Support Vector Learning*, pages 185–208. MIT Press, Cambridge.

Schölkopf, B., Smola, A., and Müller, K.-R. (1997). Kernel principal component analysis. In *International Conference on Artificial Neural Networks*, pages 583–588. Springer.

Schölkopf, B., Smola, A. J., Bach, F., and et al. (2002). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press.

Vapnik, V. N. (2000). *The Nature of Statistical Learning Theory*. Springer-Verlag, New York.

Vert, J.-P., Tsuda, K., and Schölkopf, B. (2004). A primer on kernel methods.

Wolberg, W. (1992). Breast Cancer Wisconsin (Original). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5HP4Z.

Yin, J. and Li, Q.-N. (2019). A semismooth newton method for support vector classification and regression.

Zhou, S.-s., Liu, H.-w., Zhou, L.-h., and Ye, F. (2007). Semismooth newton support vector machine. *Pattern Recognition Letters*, 28:2054–2062.

A Appendix A

A.1 Data Loader

```
from pathlib import Path
import numpy as np
import pandas as pd

from data_processor import DataProcessor
from transforms import UnitGaussianNormalizer
from utils import DictDataset


def load_breast_cancer_data(
    input_encoder: bool = False,
    output_encoder: bool = False,
    data_path: str = "data",
    saver: bool = False,
    verbose: bool = False,
):
    """loads pre- / post-processed breast cancer data

    Args
    ----
        :param input_encoder: input encoder
        :param output_encoder: output encoder
        :param data_path: path to data
        :param saver: save proecessed data, False by default
        :param verbose: verbose flag

    Returns
    -------
        :return: train_loader, test_loader, data_processor
    """

    names: list = [
        "id",
        "clump_thickness",
        "uniformity_cell_size",
        "uniformity_cell_shape",
        "marginal_adhesion",
        "single_epithelial_cell_size",
        "bare_nuclei",
        "bland_chromatin",
        "normal_nucleoli",
        "mitoses",
```

```
    "class",
]

DIR: Path = Path(__file__).parent

data: pd.DataFrame = pd.read_csv(
    DIR.joinpath(data_path, "raw/breast_cancer_wisconsin.data"),
    names=names,
    low_memory=False,
)

x: pd.DataFrame = data.drop(["id", "class"], axis=1)
y: pd.DataFrame = data["class"].map({2: -1, 4: 1}).to_frame()
del data

if not output_encoder:
    input_encoder = UnitGaussianNormalizer()
    data_processor = DataProcessor(
        input_encoder=input_encoder, output_encoder=output_encoder
    )
    x = data_processor.preprocess(x)

if input_encoder and output_encoder:
    input_encoder = UnitGaussianNormalizer()
    output_encoder = UnitGaussianNormalizer()
    data_processor = DataProcessor(
        input_encoder=input_encoder, output_encoder=output_encoder
    )
    x = data_processor.preprocess(x)
    y = data_processor.preprocess(y)

if verbose:
    print("Data loaded and processed")
    print(f"X shape: {x.shape}")
    print(f"Y shape: {y.shape}")
    print(f"X head: {x.head()}")
    print(f"Y head: {y.head()}")

split: int = int(0.8 * x.shape[0])

# train samples
x_train: pd.DataFrame = x.iloc[:split]
y_train: pd.DataFrame = y.iloc[:split]

# test samples
x_test: pd.DataFrame = x.iloc[split:]
y_test: pd.DataFrame = y.iloc[split:]
```

```
    del x, y

    if saver:
        x_train.to_csv(DIR.joinpath(data_path, "clean/train_features.csv"))
        y_train.to_csv(DIR.joinpath(data_path, "clean/train_labels.csv"))
        x_test.to_csv(DIR.joinpath(data_path, "clean/test_features.csv"))
        y_test.to_csv(DIR.joinpath(data_path, "clean/test_labels.csv"))

    train_loader = DictDataset(x_train.values, y_train.values)
    test_loader = DictDataset(x_test.values, y_test.values)

    return train_loader, test_loader, data_processor
```

A.2 Data Processor

```
import abc
import pandas as pd

from transforms import Transform


class AbstractDataProcessor(abc.ABC):
    """Data processing abstract base class for pre-
    and post-processing data for training/inference"""

    def __init__(self):
        pass

    @abc.abstractmethod
    def preprocess(self, data_frame: pd.DataFrame):
        """Preprocess data"""
        return NotImplementedError

    @abc.abstractmethod
    def postprocess(self, data_frame: pd.DataFrame):
        """Postprocess data"""
        return NotImplementedError


class DataProcessor(AbstractDataProcessor):
    """Data processor for training/inference data

    Args
    ----
        :param input_encoder: input encoder
        :param output_encoder: output encoder
        :param n_features: number of input features
```

```
    """

    def __init__(
        self,
        input_encoder: Transform = None,
        output_encoder: Transform = None,
        n_features: int = None,
    ):

        super().__init__()
        self.input_encoder = input_encoder
        self.output_encoder = output_encoder
        self.n_features = n_features

    def preprocess(self, data_frame: pd.DataFrame):
        if self.input_encoder:
            data_frame = self.input_encoder.transform(data_frame)

        return data_frame

    def postprocess(self, data_frame: pd.DataFrame):
        if self.output_encoder:
            data_frame = self.output_encoder.inverse_transform(data_frame)

        return data_frame
```

A.3 Transforms

```
import abc
import numpy as np
import pandas as pd

from typing import List, Union


class Transform(abc.ABC):
    """Transform base class"""

    def __init__(self):
        super().__init__()

    @abc.abstractmethod
    def transform(self, data_frame: pd.DataFrame):
        """Forward transform"""
        pass

    @abc.abstractmethod
```

```
    def inverse_transform(self, data_frame: pd.DataFrame):
        """Inverse transform"""
        pass


class Composite(Transform):
    """Composite transform for multiprocess data transforms

    Input is a list of transformations, must be instantiations
    of the Transform class.

    Args
    ----
        :param transforms: transformations pipeline
    """

    def __init__(self, transforms: List[Transform]):
        super().__init__()
        self.transforms = transforms

    def transform(self, data_frame: pd.DataFrame) -> pd.DataFrame:
        """Forward transfroms"""
        for tform in self.transforms:
            data_frame = tform.transform(data_frame)

        return data_frame

    def inverse_transform(self, data_frame: pd.DataFrame) -> pd.DataFrame:
        """Inverse transforms"""
        for tform in reversed(self.transforms):
            data_frame = tform.inverse_transform(data_frame)

        return data_frame


class UnitGaussianNormalizer(Transform):
    """Normalizes data to unit Gaussian

    Args
    ----
        :param dims: dimensional discriminator; since dataframe
            dims is a list of strings corresponding to columns
    """

    def __init__(self, dims: list = None):
        super().__init__()
        self.dims = dims
```

```
        self.mean: dict = {}
        self.std: dict = {}

    def transform(self, data_frame: pd.DataFrame) -> pd.DataFrame:
        """Forward transform"""
        if not self.dims:
            self.dims = data_frame.columns
        for dim in self.dims:
            values: np.array = np.array(data_frame[dim].values)
            self.mean[dim] = np.mean(values)
            self.std[dim] = np.std(values)
            data_frame[dim] = (values - self.mean[dim]) / self.std[dim]
        return data_frame

    def inverse_transform(self, data_frame: pd.DataFrame) -> pd.DataFrame:
        """Inverse transform"""
        for dim in self.dims:
            values: np.array = np.array(data_frame[dim].values)
            data_frame[dim] = values * self.std[dim] + self.mean[dim]
        return data_frame
```

A.4 Semismooth Newton Optimiser for SVM

```
import abc
import numpy as np
import pandas as pd

from utils import Callback
from handlers import F


class Optimizer(abc.ABC):
    """Abstract base optimizer class"""

    def __init__(self):
        super().__init__()
        pass

    @abc.abstractmethod
    def optimize(self):
        """Optimize function"""
        return NotImplementedError


class svmOptimiser(Optimizer):
    """Semi-smooth Newton optimiser for SVM"""
```

```
def __init__(
    self,
    x: np.array,
    y: np.array,
    sigma: float = 1e-2,
    delta: float = 1e-6,
    eta: float = 0.05,
    kappa: float = 0.3,
    C: float = None,
    rho: float = 0.9,
    maxIter: int = 1000,
    w_init: np.array = False,
):
    super().__init__()

    # data
    self.x = x
    self.y = y

    # parameters
    self.sigma = sigma
    self.delta = delta
    self.eta = eta
    self.kappa = kappa
    self.C = C
    self.rho = rho
    self.maxIter = maxIter
    self.stop_condition = True
    self.sIter = 0
    self.cgIter = 0

    if not w_init:
        w_init = np.ones(x.shape[1])

    self.w = w_init
    self.data = pd.DataFrame(
        columns=
        [
            "Iteration",
            "Objective",
            "Gradient Norm",
            "Active Set Size",
            "w"
        ]
    )

def optimize(
```

```
    self,
    f: F,
    callback: Callback = None,
):
    """implements semismooth newton optimization"""
    while self.stop_condition and self.sIter < self.maxIter:
        self.stop_condition: bool = (
            np.linalg.norm(
                f.gradient(self.w, self.x, self.y, self.C), ord=2
            ) > self.delta
        )

        if self.sIter > 1 and False:
            if callback:
                self.stop_condition: bool = (
                    self.data["Active Set Size"].iloc[-2]
                    - self.data["Active Set Size"].iloc[-1]
                    > 0
                )

        g: np.array = f.gradient(self.w, self.x, self.y, self.C)
        V: np.array = f.V(
            self.w,
            self.x,
            self.y,
            f.active_set(
                self.w, self.x, self.y
                ),
            self.C
        )
        d: np.array = np.ones(self.w.shape[0])
        r: np.array = -g - np.dot(V, d)
        p: np.array = r
        psi: float = np.min([self.eta, self.kappa * np.linalg.norm(g)])

        cg_stop_condition: bool = True
        cg_maxIter: int = 200
        j = 0
        while cg_stop_condition and j < cg_maxIter:

            if psi * np.linalg.norm(g) > np.linalg.norm(r):
                cg_stop_condition = False

            alpha: float = np.dot(r.T, r) / np.dot(p.T, np.dot(V, p))
            d = d + alpha * p  # update
            r_ = r
            r = r - alpha * np.dot(V, p)
```

```
        if psi * np.linalg.norm(g) > np.linalg.norm(r):
            cg_stop_condition = False

        beta: float = np.dot(r.T, r - r_) / np.dot((r - r_).T, p)
        p = r + beta * p  # update

        j += 1

    # cg iter end
    self.cgIter += j

    alpha = self.rho
    k = 0
    while f.objective(
    self.w + alpha * d, self.x, self.y, self.C
    ) > f.objective(
        self.w, self.x, self.y, self.C
    ) + self.sigma * alpha * np.dot(g.T, d):
        k += 1
        alpha = alpha**k

    # line search end

    self.w = self.w + alpha * d

    if callback:
        callback.update_state_dict(
            active_set_size=f.active_set(
                self.w, self.x, self.y
            ).shape[0],
            objective=f.objective(
                self.w, self.x, self.y, self.C
            ),
            w=self.w,
            iter=self.sIter,
            gradient_norm=np.linalg.norm(
                f.gradient(self.w, self.x, self.y, self.C)
            ),
        )

    self.data.loc[self.sIter, :] = [
        self.sIter,
        f.objective(self.w, self.x, self.y, self.C),
        np.linalg.norm(g),
        len(f.active_set(self.w, self.x, self.y)),
        self.w,
```

```
            ]

            self.sIter += 1

        return self.data, self.w, self.sIter, self.cgIter
```

A.5 Function Pointers / Handlers

We include the linear formulation in **Definition 1.3**, and the non-linear with both Polynomial Kernel (even though it is not covered in the report), and RBF kernel as in **Definition 1.4**.

```
import numpy as np
import pandas as pd


class F(object):
    """function handler for SVM optimisation"""

    def __init__(self):
        pass

    def objective(
        w: np.ndarray, x: np.ndarray, y: np.ndarray, C: float
    ) -> float:
        """objective function."""
        return 0.5 * np.linalg.norm(w, ord=2) ** 2 + C * np.sum(
            np.maximum(0, 1 - y * np.dot(x, w)) ** 2
        )

    def gradient(
        w: np.ndarray, x: np.ndarray, y: np.ndarray, C: float
    ) -> np.ndarray:
        """gradient of the objective function."""
        return w - 2 * C * np.sum(
            np.multiply(
                np.maximum(
                    0, 1 - y * np.dot(x, w)
                ) * y, x.T).T, axis=0
        )

    def active_set(
        w: np.ndarray, x: np.ndarray, y: np.ndarray
    ) -> list:
        """active set function"""
        return np.where(1 - y * np.dot(x, w) > 0)[0]
```

```
    def V(
        w: np.ndarray, x: np.ndarray, y: np.ndarray, active: list, C: float
    ) -> np.ndarray:
        """generalised jacobian space, V"""
        return np.eye(len(w)) + 2 * C * x[active].T @ x[active]


class FPolyKernel(object):
    """function handler for SVM optimisation"""

    def __init__(self,
    gamma: float = 1 / 9,
    coef0: int = 0,
    degree: int = 2
):
        self.gamma = gamma
        self.coef0 = coef0
        self.degree = degree

    def poly_kernel(self, x1, x2):
        """Polynomial kernel function"""
        return (self.gamma * np.dot(x1, x2) + self.coef0) ** self.degree

    def objective(
        self,
        w: np.ndarray,
        x: np.ndarray,
        y: np.ndarray,
        C: float,
    ) -> float:
        """objective function."""
        return 0.5 * np.linalg.norm(w, ord=2) ** 2 + C * np.sum(
            np.maximum(
                0, 1 - y * np.array([self.poly_kernel(xi, w) for xi in x])
                ) ** 2
        )

    def gradient(
        self, w: np.ndarray, x: np.ndarray, y: np.ndarray, C: float
    ) -> np.ndarray:
        """gradient of the objective function."""
        return w - 2 * C * np.sum(
            np.multiply(
                np.maximum(
                    0, 1 - y * np.array(
                        [self.poly_kernel(xi, w) for xi in x]
                    )
```

```
                )* y,
                x.T,
            ).T,
            axis=0,
        )

    def active_set(self, w: np.ndarray, x: np.ndarray, y: np.ndarray) -> list:
        """active set function"""
        return np.where(
            1 - y * np.array([self.poly_kernel(xi, w) for xi in x]
        ) > 0)[0]

    def V(
        self,
        w: np.ndarray,
        x: np.ndarray,
        y: np.ndarray,
        active: list,
        C: float
    ) -> np.ndarray:
        """generalised jacobian space, V"""
        return np.eye(len(w)) + 2 * C * np.array(
            [self.poly_kernel(xi, w) for xi in x[active]]
        ).T @ np.array([self.poly_kernel(xi, w) for xi in x[active]])


class FGaussianKernel(object):
    """function handler for SVM optimisation"""

    def __init__(self, gamma: float = 1 / 9):
        self.gamma = gamma

    def rbf_kernel(self, x1, x2):
        """RBF kernel function"""
        return np.exp(-self.gamma * np.linalg.norm(x1 - x2) ** 2)

    def objective(
        self,
        w: np.ndarray,
        x: np.ndarray,
        y: np.ndarray,
        C: float,
    ) -> float:
        """objective function."""
        return 0.5 * np.linalg.norm(w, ord=2) ** 2 + C * np.sum(
            np.maximum(
                0, 1 - y * np.array(
```

```
                    [self.rbf_kernel(xi, w) for xi in x]
                )) ** 2
        )

    def gradient(
        self, w: np.ndarray, x: np.ndarray, y: np.ndarray, C: float
    ) -> np.ndarray:
        """gradient of the objective function."""
        return w - 2 * C * np.sum(
            np.multiply(
                np.maximum(0, 1 - y * np.array(
                    [self.rbf_kernel(xi, w) for xi in x]
                    )
                )* y,
                x.T,
            ).T,
            axis=0,
        )

    def active_set(
        self,
        w: np.ndarray,
        x: np.ndarray,
        y: np.ndarray
    ) -> list:
        """active set function"""
        return np.where(
            1 - y * np.array([self.rbf_kernel(xi, w) for xi in x]) > 0
        )[0]

    def V(
        self,
        w: np.ndarray,
        x: np.ndarray,
        y: np.ndarray,
        active: list,
        C: float
    ) -> np.ndarray:
        """generalised jacobian space, V"""
        return np.eye(len(w)) + 2 * C * np.array(
            [self.rbf_kernel(xi, w) for xi in x[active]]
        ).T @ np.array([self.rbf_kernel(xi, w) for xi in x[active]])
```

A.6 Plot functions

```
import numpy as np
import pandas as pd
```

```python
import matplotlib.pyplot as plt

from utils import Callback
from handlers import F


plt.rcParams.update(
    {
        "text.usetex": True,  # enable LaTex formatting
        "grid.linestyle": "dotted",  # dotted grid background
        "figure.figsize": (7, 5),  # figure size (width, height) in inches
        "font.family": "Computer Modern Roman",  # font (default LaTex font)
        "text.latex.preamble" : r"\usepackage{amsmath}",
    }
)


def q_convergence(data, q: int = 1) -> plt.plot:
    """Convergence plot for q"""
    iters = data["Iteration"]
    nerr: list = []
    w = data["w"].tolist()  # Convert the 'w' column to a list
    for i in range(1, len(w)):
        nerr.append(
            np.linalg.norm(np.array(w[i]) - np.array(w[-1]))
            / (np.linalg.norm(np.array(w[i - 1]) - np.array(w[-1]))) ** q
        )

    plt.plot(iters[1:], nerr, "m")
    plt.plot(iters[1:], nerr, "xk")
    plt.xlabel(r"Iter, $k$")
    plt.ylabel(
        r"$\frac{\|\boldsymbol{w}_{k}
        - \boldsymbol{w}^*\|}{\|\boldsymbol{w}_{k-1} -
        \boldsymbol{w}^*\|^q}$"
    )
    plt.title(f"Q-convergence, $q = {q}$")
    plt.grid()
    plt.show()


def exponential_convergence(data) -> plt.plot:
    """Exponential convergence plot"""
    iters = data["Iteration"]
    nerr: list = []
    w = data["w"].tolist()
    for i in range(1, len(w)):
```

```
            nerr.append(np.linalg.norm(np.array(w[i]) - np.array(w[-1]), ord=1))

        plt.plot(iters[1:], nerr, "m")
        plt.plot(iters[1:], nerr, "xk")
        plt.yscale("log")
        plt.xlabel(r"Iter, $k$")
        plt.ylabel(r"$\|\boldsymbol{w}_{k} - \boldsymbol{w}^*\|$")
        plt.title("Exponential Convergence")
        plt.grid()
        plt.show()


    def algebraic_convergence(data) -> plt.plot:
        """Algebraic convergence plot"""
        iters = data["Iteration"]
        nerr: list = []
        w = data["w"].tolist()
        for i in range(1, len(w)):
            nerr.append(np.linalg.norm(np.array(w[i]) - np.array(w[-1]), ord=2))

        plt.plot(iters[1:], nerr, "m")
        plt.plot(iters[1:], nerr, "xk")
        plt.yscale("log")
        plt.xscale("log")
        plt.xlabel(r"Iter, $k$")
        plt.ylabel(r"$\|\boldsymbol{w}_{k} - \boldsymbol{w}^*\|$")
        plt.title("Algebraic Convergence")
        plt.grid()
        plt.show()


    def objective_error(data) -> plt.plot:
        """Objective error plot"""
        iters = data["Iteration"]
        nerr: list = []
        obj = data["Objective"].tolist()
        for i in range(1, len(obj)):
            nerr.append(np.abs(obj[i] - obj[-1]))

        plt.plot(iters[1:], nerr, "m")
        plt.plot(iters[1:], nerr, "xk")
        plt.yscale("log")
        plt.xlabel(r"Iter, $k$")
        plt.ylabel(r"$|f(\boldsymbol{w}_{k}) - f(\boldsymbol{w}^*)|$")
        plt.title("Objective Error")
        plt.grid()
        plt.show()
```

```
def grad_norm(data) -> plt.plot:
    """Gradient norm plot"""
    iters = data["Iteration"]
    norm: list = []
    grad = data["Gradient Norm"].tolist()
    plt.plot(iters, grad, "m")
    plt.plot(iters, grad, "xk")
    plt.yscale("log")
    plt.xlabel(r"Iter, $k$")
    plt.ylabel(r"$\| \nabla f(\boldsymbol{w}_{k})\|$")
    plt.title("Gradient Norm Error")
    plt.grid()
    plt.show()
```