

Subspace Newton Method with Sparse SVMs (NSSVM)

Student ID: 23205123

Date: 19 April 2024

Words: 2994

1 SVM Classification and underlying optimization problem

1.1 Binary Classification Problem

Binary classification in machine learning is a problem where the objective is to learn a classification rule that assigns one of two possible discrete labels to each data instance in a dataset. Given a set of training points $\{(x_i, y_i)_{i=1}^m\}$, where each $x_i \in \mathbb{R}^n$ is a feature vector and $y_i \in -1, 1$ is the binary label. The goal is to find a function $f : \mathbb{R}^n \rightarrow \{-1, 1\}$, that can accurately predict label of new, unseen data [1]. Task is to train a hyperplane $\langle w, x \rangle + b = 0$ with variable $w \in \mathbb{R}^n$ and bias $b \in \mathbb{R}$ to be approximated based on training dataset on both synthetic and real dataset. Synthetic and real dataset generation is as detailed in Appendix 4.2 and Zhou [2].

1.2 Sparse SVMs

To obtain an optimal hyperplane, a conventional approach to solve this is soft-margin SVM optimization for inseparable training datasets in the input space. Extensive research has been conducted on the design of loss functions ℓ for soft-margin SVMs [3, 4, 5, 6]. This gives rise to the Hinge loss function. To address this problem, the dual kernel-based SVM is usually used. However, solutions to the dual kernel-based SVM optimization are not sufficiently sparse. Consequently, extensive research follows for investigating methods for data reduction on Sparse SVMs [7, 8, 9]. Sparse SVMs exploit intrinsic sparsity in the data by imposing an ℓ_1 -norm regularization in the loss function, which tends to produce a solution with many 0 coefficients, effectively reducing number of support vectors. This aligns with concept that only a subset of training samples, support vectors have significant impact on decision boundary. Reduction in number of support vectors decreases model complexity, which translates to less memory usage and faster prediction times.

In this problem, we focus on this following soft-margin SVM, which is the foundation of the the

sparsity constrained kernel based optimization problem formulation in Equation 1, detailed in Section 1.3:

$$\min_{w \in \mathbb{R}^n, b \in \mathbb{R}} \frac{1}{2} \|w\|^2 + \sum_{i=1}^m \ell_{cC} [1 - y_i (\langle w, x_i \rangle + b)] \quad (1)$$

where the dual problem of Equation 1 is:

$$\min_{\alpha \in \mathbb{R}^m} D(\alpha) := d(\alpha) + \sum_{i=1}^m h_{cC}(\alpha_i), \text{ s.t. } \sum_{i=1}^m \alpha_i y_i = 0 \quad (2)$$

where the squared Hinge loss, h_{cC} is described by Equation 3 below. Based on Zhou [2]'s (Theorem 2.1), the primal loss is reduced to squared Hinge loss from hinge loss when $c \rightarrow 0$:

$$h_{cC}(t) := \begin{cases} \frac{t^2}{2C}, & t \geq 0 \\ \frac{t^2}{2c}, & t < 0 \end{cases} \quad (3)$$

1.3 Sparsity Constrained Kernel-based Optimization Problem

Detailed by Zhou [2] and extended research on sparse SVMs, the dual problem of the soft-margin SVM optimization becomes:

$$\min_{\alpha \in \mathbb{R}^m} \frac{1}{2} \|\mathbf{Q}\alpha\|^2 + \sum_{i=1}^m h_{c,C}(\alpha_i) - \langle \mathbf{1}, \alpha \rangle =: \mathbf{D}(\alpha),$$

s.t. $\langle \alpha, \mathbf{y} \rangle = 0, \|\alpha\|_0 \leq s,$

where an user-aligned integer $s \in \mathbb{N}_m$ is less than m and is called the sparsity level and $\|\alpha\|_0$ is number of non-zero elements of α . Here, as shown above in Equation 3:

$$h_{c,C}(t) := \frac{t^2}{2} [C\rho(t \geq 0) + c\rho(t < 0)]^{-1} \quad (4)$$

where $C \geq c > 0$ are two parameters and $\rho(t \in S)$ is an indicator function. When $t \in S$, this returns 1 and 0 otherwise. Based on Equation 3 and 4, this implies a given penalty of $1/C$ for $\alpha_i \geq 0$ and $1/c$ for $\alpha_i < 0$. If $c \rightarrow 0$, then $\alpha_i \geq 0, i \in \mathbb{N}_m$. We will be working with a type of dual formulation of the soft-margin optimization formulation with sparse SVMs.

1.3.1 Properties of Optimization Problem

Compared to the conventional dual problem of soft-margin SVM optimization, which is the quadratic kernel SVM optimization shown here:

$$\begin{aligned} \min_{\alpha \in \mathbb{R}^m} & \frac{1}{2} \|Q\alpha\|^2 - \langle \alpha, \mathbf{1} \rangle \\ \text{s.t. } & \langle \alpha, \mathbf{y} \rangle = 0, \quad 0 \leq \alpha_i \leq C, i \in \mathbb{N}_m \end{aligned} \quad (5)$$

where $Q := [y_1 \mathbf{x}_1, \dots, y_m \mathbf{x}_m] \in \mathbb{R}^{n \times m}$, $\mathbf{y} := (y_1, \dots, y_m)^\top \in \mathbb{R}^m$, $\mathbf{1} := (1, \dots, 1)^\top \in \mathbb{R}^m$ and $\mathbb{N}_m := \{1, \dots, m\}$. For the quadratic kernel optimization problems, it is good to understand that the challenge that it poses is that the matrix $Q^T Q$ has $m \times m$ -order, which makes it challenging to train on 10^6 data and incurs large computational cost [9]. Complexity of computing this kernel is around $O(m^2 n)$. Therefore, dimensionality reduction has been attempted by researchers to train SVMs on smaller datasets [10]. This is also why we have utilized this dual problem coined the sparsity constrained kernel SVM optimization problem.

Our problem in Section 1.3, has 3 substantial advantages:

1. The objective function exhibits strong convexity, therefore, this ensures the existence of optimal solutions, and might be unique under mild conditions, as proved by Zhou [2]. The objective function shown in Equation 5 is convex, but not strongly convex when $n \geq m$ as $Q \in \mathbb{R}^{n \times m}$. This indicates numerous optimal solutions due to dimensionality of space $Q \in \mathbb{R}^{n \times m}$.
2. As there is no longer fixed constraints this simplifies the computational process, making it more feasible. The bounded constraints of Equation 5 might lead to significant increase in computational demands.
3. Sparsity constraint $\|\alpha\|_0 \leq s$ which specifies vector α should have at most s non-zero components. This implies that model is expected to use no more than s support vectors, such as represented by the Representer Theorem:

$$w = Q\alpha = \sum_{i=1}^m \alpha_i y_i x_i$$

2 Optimization method and convergence theory

2.1 Choice of Optimization Method

The appropriate method chosen for solving the stationary equations in 6, is subspace Newton method.

The justifications for this choice as laid below in the consequent subsections. Firstly, as demonstrated by [11, 12, 13], this Newton method variant, is not only computationally efficient, but also ensures rapid convergence. A good thing to note is this method surpasses the conventionally observed locally quadratic convergence property of Newton methods by guaranteeing convergence to an equilibrium point in one iteration, provided the initial guess η -stationary point is sufficiently close as outlined in Zhou [2]'s Theorem 3.1. The method also address the issue of determining the appropriate number of support vectors through s which is the sparsity level. It is worth mentioning that although deciding arbitrary number s is challenging, the method includes an innovative way to adjust s , called sparsity level tuning. As shown by the author, CPU time for large scale datasets with 10^5 size, the method took approximately 0.04 seconds, faster than Lagrangian SVMs [14] and Fuzzy SVMs [15] (both 0.10s) with a standardised synthetic benchmark.

It is stated by the author that while the exact proximity of starting point to an η -stationary point is not clear, the method succeeds across various starting points which suggests a degree of insensitivity to these initial conditions. Once the method/ algorithm is in the neighbourhood of an η -stationary point, it is capable of determining local, and potentially global minimum depending on sparsity level s and relation of η^* to the penalty parameter C . Thus, algorithm's design not only facilitates finding local minima but also under certain parameters and conditions, ensures a global minimizer. Notably, Zhou [2] has shown theoretical one-step convergence result as defined in his paper and in Section 2.5.

Furthermore, as this is a Newton method derivative, multiple attempts at subspace Newton method have theoretically proven that this subspace Newton method has global convergence under certain assumptions and the convergence rate is the same as of the full regularized Newton method or traditional Newton Methods [11, 12, 13]. For example, Fuji, Poirion and Takeda [11] has theoretically and empirically proven that Randomized Subspace Newton method (RSN) has also shown to achieve a global linear convergence for strongly convex f . Additionally, subspace Newton methods are expected to be highly computationally efficient compared to traditional Newton method, since it does not require computation of the full Hessian inverse.

2.2 Groundwork: List of Notations

Table 1: Glossary of Symbols

Notation	Details
$[m]$	Set containing sequential integers from 1 to m .
$ T $	Cardinality or the count of elements within set T .
T	Complementary set consisting of indices not included in a specific index set.
α_T	A vector formed by selecting elements of α corresponding to indices in set T .
$ \alpha $	Vector with absolute values of elements from α .
$\ \alpha\ _{[s]}$	The s -th highest value within the vector α .
$\text{supp}(\alpha)$	Set containing indices where elements of α are non-zero.
y	A vector of m labels or classes associated with the samples.
X	Data matrix with m samples as rows and n features as columns.
Q	Matrix obtained by element-wise multiplication of each label in y with the corresponding row in X .
Q_T	A matrix with columns chosen from Q based on indices in set T .
$Q_{\Gamma, T}$	A matrix with rows and columns of Q_{Γ} indexed by set T .
I	Identity matrix of appropriate size.
P	Resultant matrix after adding the inverse of the regularization constant C times the identity matrix to $Q^T Q$.
$\mathbf{1}$	Vector consisting entirely of ones, with the dimension context-dependent.
$N(\alpha, \delta)$	The δ -neighborhood around vector α , which includes all vectors within a distance of δ .
$\{u\}$	Set containing the vector u .

2.3 Subspace Newton Method

Compared to the traditional Newton method, which computes the Newton step in the entire space of variables, which can be computationally intensive, the subspace Newton method selects a lower-dimensional subspace in which to perform the optimization. The method is based on finding an η -stationary point for the optimization task. A point $z^* = (p^*, b^*)$ (detailed below) with p^* being a solution within the sparsity constraints of the problem is termed an η -stationary point if it satisfies a set of conditions characterized by system of equations $F(z^*; T^*) = 0$ below. In essence,

the η -stationary point brings b^* , the Lagrangian multiplier which acts as a bias in this SVM formulation. By satisfying the η -stationary point conditions, one can find solutions to both primal and dual forms of the problem simultaneously.

This diverges from the traditional Newton methods by focusing on the subspace of problem defined by the non-zero components of p^* , constrained by the sparsity level.

Theorem 1 [2]: A point z^* is an η -stationary point of optimization problem for some $\eta > 0$ if and only if there is a $T_* \in \mathbb{T}_s(\alpha^* - \eta g(z^*))$ such that:

$$F(z^*; T_*) = \begin{bmatrix} g_{T_*}(\alpha^*) \\ \alpha_{T_*}^* \\ \langle \alpha_{T_*}^*, y_{T_*} \rangle \end{bmatrix} = \begin{bmatrix} H_{T_*}(\alpha^*) \alpha_{T_*}^* - 1 + y_{T_*} \mu^* \\ \alpha_{T_*}^* \\ \langle \alpha_{T_*}^*, y_{T_*} \rangle \end{bmatrix} = 0 \quad (6)$$

The notion of an η -stationary point in optimization is captured by a set of equations reflecting the balance between the objective function's gradient and the constraints. For a point to be η -stationary, it must satisfy these Equations 6 within a particular subspace. Hence, with proof of conditions in Zhou [2](Section 2.11), this enables us to utilize Newton method. Additionally, if point z is η -stationary point of Equation 3, the dual SVM problem, for some $\eta > 0$, then $T \in \mathbb{T}_s(\alpha - \eta g(z))$, the Jacobian matrix of this system:

$$\nabla F(z; T) = \begin{bmatrix} H_T(\alpha) & 0 & y_T \\ 0 & I & 0 \\ y_T^T & 0 & 0 \end{bmatrix} \quad (7)$$

retains non-singularity, ensuring the robustness of the Newton method's iterative steps. This is due to congruence of $\nabla F(z; T)$ to a non-singular matrix as $H_T(\alpha)$ is positive semi-definite and $y_T^T (H_T(\alpha))^{-1} y_T > 0$.

$$\begin{bmatrix} H_T(\alpha) & 0 & y_T \\ 0 & I & 0 \\ 0 & 0 & y_T^T (H_T(\alpha))^{-1} y_T \end{bmatrix}$$

This attribute arises from the positive semi-definite nature of the Hessian of the objective function, rendering the convergence mechanism of the Newton method both reliable and effective. With z_k defined as

$$\begin{bmatrix} \alpha^k \\ \mu^k \end{bmatrix}$$

, and approximate solution to stationary equations 6, we gain:

$$\begin{bmatrix} H_{T_k}(\alpha^k) & 0 & Y_{T_k} \\ 0 & I & 0 \\ Y_{T_k}^T & 0 & 0 \end{bmatrix} \begin{bmatrix} d_{T_k}^k \\ d_{\bar{T}_k}^k \\ d_{m+1}^k \end{bmatrix} = - \begin{bmatrix} g_{T_k}(z^k) \\ \alpha_{\bar{T}_k}^k \\ (\alpha_{T_k}^k, Y_{T_k}) \end{bmatrix}.$$

With the direction, Newton step size is taken and brings out:

$$z^{k+1} = z^k + d^k = \begin{bmatrix} \alpha_{T_k}^k \\ \alpha_{\bar{T}_k}^k \\ \mu^k \end{bmatrix} + \begin{bmatrix} d_{T_k}^k \\ d_{\bar{T}_k}^k \\ d_{m+1}^k \end{bmatrix} = \begin{bmatrix} \alpha_{T_k}^k + d_{T_k}^k \\ 0 \\ \mu^k + d_{m+1}^k \end{bmatrix}.$$

Therefore, the algorithm is summarised in the pseudocode below.

Algorithm [2] NSSVM: Subspace Newton method with adaptively tuning s for sparse SVMs

Input: Give parameters $C, c > 0, \eta > 0, r > 1$, MaxACC = 0, $s_0 \in \mathbb{N}_m$, Tol and MaxIt.

Initialize: z^0 , pick $T_0 \in T_{s_0}(\alpha^0 - \eta g(z^0))$ and set $k := 0$.

while $(\|F(z^k; T_k)\| \geq \text{Tol or } |\text{ACC}(\alpha^k) - \text{MaxACC}| > 10^{-4})$ and $(k < \text{MaxIt})$ **do**

 Update d^k by solving (3.2).

 Update z^{k+1} by (3.3).

 Update MaxACC = $\max\{\text{ACC}(\alpha^1), \dots, \text{ACC}(\alpha^{k-1})\}$.

 Update $s_{k+1} = rs_k$ if k is a multiple of 10 and $s_{k+1} = s_k$ otherwise.

 Update $T_{k+1} \in T_{s_{k+1}}(\alpha^{k+1} - \eta g(z^{k+1}))$ and set $k := k + 1$.

end while

return the solution z^k .

2.4 Local and Global Convergence Properties

Zhou [2] claims that subspace Newton method demonstrates one step convergence to a local minimizer under certain conditions as stated above. The proofs in his paper implies that under the right conditions, one might expect a type of convergence that is at least Q-linear due to iterative improvement at each step, possibly Q-superlinear if updates align well with the conditions of the η -stationary point, leading to rapid improvement towards optimum. Local quadratic convergence might also be inferred, as this is a second-order method (using second derivatives), like the traditional Newton method.

Based on Zhou [2]'s observations for the objective function here:

$$D(\alpha) := \frac{1}{2} \|Q\alpha\|^2 + \sum_{i=1}^m h_{c,C}(u_i) - \langle 1, \alpha \rangle$$

The gradient $\nabla D(\alpha)$ and Hessian $H(\alpha)$ are:

$$\nabla D(\alpha) := H(\alpha)\alpha - 1$$

$$H(\alpha) := Q^T Q + E(\alpha)$$

where $E(\alpha)$ is diagonal matrix, as shown in []:

$$E_{ii}(\alpha) := (E(\alpha))_{ii} = \begin{cases} \frac{1}{C}, & \alpha_i \geq 0, \\ \frac{1}{c}, & \alpha_i < 0. \end{cases}$$

By Mercer's Theorem, Hessian matrix is positive definite for any $\alpha \in \mathbb{R}^m$. Let $H(\alpha) \succeq \frac{1}{C} Q^T Q + I$ where \succeq denotes the matrix being positive semi-definite, indicating that $D(\alpha)$ is a strongly convex function and therefore inherits certain properties useful for optimization. Specifically, for any $\alpha, \alpha' \in \mathbb{R}^m$, where $t \in [0, 1]$ and by the mean value theorem [16], we have:

$$\begin{aligned} D(\alpha) &\geq D(\alpha') + \langle \nabla D(\alpha'), \alpha - \alpha' \rangle \\ &\quad + \frac{1}{2} \langle \alpha - \alpha', H(\bar{\alpha})(\alpha - \alpha') \rangle \end{aligned} \quad (8)$$

with $\bar{\alpha} = \alpha + t(\alpha' - \alpha)$ for some $t \in [0, 1]$ and $\bar{\alpha}$ guaranteed by the mean value theorem. We define z as a concatenation of α and the Lagrange multipliers μ , and similarly for z^* and z^k . We then denote functions $g(z)$ and $g_T(z)$, corresponding to the gradient of D and the sub-vector of g , respectively, with $H_T(\alpha)$ representing the sub-principal matrix of $H(\alpha)$ indexed by T .

Based on Zhou [2]'s Theorem 2.4 laid out below, both local and global convergence can be expected for this sparsity constrained optimization problem:

Theorem 2.4: Given the problem in Equation 3 and a point α^* that fulfills $\|\alpha^*\|_0 \leq s$ and $(\alpha^*, y) = 0$, the following statements are proposed:

- (a) A point α^* becomes a local minimizer if it is an η -stationary point for a particular $\eta > 0$.
- (b) α^* is regarded as a local minimizer if it is an η -stationary point for a given $\eta > 0$ when $\|\alpha^*\|_0 < s$, or if the specific condition $\|\alpha^*\|_0 = s$ holds with η defined as

$$\eta^* = \frac{\|\alpha^*\|_{[s]}}{2|(H(\alpha^*)\alpha^* - 1)|} > 0.$$

- (c) If $\|\alpha^*\|_0 < s$, the points of local minimizer, global minimizer, and η -stationary are coinciding and unique.

- (d) When $\|\alpha^*\|_0 = s$, and α^* is an η -stationary point for some η meeting the condition

$$\left[\frac{1}{C} - \frac{1}{\eta} \right] I + Q^T Q \preceq 0,$$

it is then also a global minimizer. Furthermore, this minimizer is unique if the strict inequality is satisfied in the above condition.

Proof of Theorem 2.4: In order to prove this, we need this lemma:

Lemma 1: A point α^* is a local minimizer if KKT condition are fulfilled such that

$$\begin{cases} g_{s_*}(\mathbf{z}^*) = 0, \\ \langle \alpha^*, \mathbf{y} \rangle = 0, \\ \|\alpha^*\|_0 = s, \end{cases} \quad \text{or} \quad \begin{cases} g(\mathbf{z}^*) = 0 \\ \langle \alpha^*, \mathbf{y} \rangle = 0 \\ \|\alpha^*\|_0 < s \end{cases}$$

It is shown from Pan, Xiu and Fan [17], that these conditions hold for the sparsity constrained problem in Equation 3.

Proof: We prove that the KKT point α^* is unique if $\|\alpha^*\|_0 < s$. If there is an other KKT point $\alpha \neq \alpha^*$, then the strong convexity of $D(\cdot)$ gives rise to:

$$\begin{aligned} D(\alpha) &\geq D(\alpha^*) + \frac{1}{2} \langle \alpha - \alpha^*, P(\alpha - \alpha^*) \rangle \\ &\quad + \langle \nabla D(\alpha^*), \alpha - \alpha^* \rangle \\ &= D(\alpha^*) + \frac{1}{2} \langle \alpha - \alpha^*, P(\alpha - \alpha^*) \rangle \\ &\quad + \langle g(\mathbf{z}^*) - \mathbf{y}\mu^*, \alpha - \alpha^* \rangle \\ &= D(\alpha^*) + \frac{1}{2} \langle \alpha - \alpha^*, P(\alpha - \alpha^*) \rangle \\ &\quad - \langle \mathbf{y}\mu^*, \alpha - \alpha^* \rangle \\ &= D(\alpha^*) + \frac{1}{2} \langle \alpha - \alpha^*, P(\alpha - \alpha^*) \rangle \\ &> D(\alpha^*), \end{aligned}$$

where when global minimizers and local minimizers coincide, making α^* unique. This is because dual function $D(\alpha)$ in optimization is strongly convex. This further supports the uniqueness of the KKT point under these conditions by showing if another KKT point existed, this would lead to a contradiction with strong convexity of $D(\alpha)$. If support of $\alpha^* = s$ and α^* is η -stationary point for some η , the Jacobian matrix will be PSD, thereby implying α^* is unique and a global minimizer.

Based on this proof and KKT optimality conditions for sparse nonlinear optimization [17], [2] discusses the proof of the statements where it reaches local and global convergence in Section 2.1.1 and 2.1.2 in his paper.

2.5 Theoretical Local Convergence Rates

It is theoretically proven that if starting point is chosen within local region, the NSSVM will take on step to terminate. This is based on Zhou [2]'s Theorem 3.1 stated below:

Theorem 3.1 (One step convergence)

Let \mathbf{z}^* be an η -stationary point of Equation 3 for some $0 < \eta < \eta^*$, and η^* and δ^* be given by Appendix 4.1.3. Let $\{\mathbf{z}^k\}$ be generated by NSSVM. There should exists at least one k such that $\mathbf{z}^k \in U(\mathbf{z}^*, \delta^*)$. Therefore we have:

$$\mathbf{z}^{k+1} = \mathbf{z}^*, \quad \left\| F(\mathbf{z}^{k+1}, T_{k+1}) \right\| = 0.$$

Namely, NSSVM terminates at the $(k+1)$ step. This proof can be seen in Appendix 4.1.2. In summary, the proof considers a point \mathbf{z}^k which is a combination of optimal point \mathbf{z}^* and difference scaled by factor t . The distance between current point \mathbf{z}^k and optimal point is shown to be decreasing. The proof establishes that the derivative function evaluated at two points related to optimal solution remain the same. With the mean value theorem, the proof shows that there is an intermediate point between the current and optimal points that connects the functions and its gradient. It shows that \mathbf{z}^* can be expressed by the current point, gradient and difference between function values at current and optimal points. This reinforces the idea that the sequence converges to an optimal point. With Newton-type method, where the function is convex, twice differentiable and Lipschitz continuous, I suspect that it can either have one-step convergence or local quadratic convergence, which is stronger than super-linear convergence. The formal theorem and property of this is as shown in Theorem A below. It has also been proven by that this Newton method derivative [11][13] follows typical Newton method convergence behaviour. Therefore:

Theorem A [18]: Local Convergence Rate of Newton's Method

Assume $F: \mathbb{R}^n \rightarrow \mathbb{R}^n$ is continuously differentiable and $x^* \in \mathbb{R}^n$ is a root of F , that is, $F(x^*) = 0$ such that $F'(x^*)$ is non-singular. Then:

- a) There exists $\delta > 0$ such that if $\|x(0) - x^*\| < \delta$ then Newton's method is well defined and

$$\lim_{k \rightarrow \infty} \frac{\|x^{(k+1)} - x^*\|}{\|x^{(k)} - x^*\|} = 0.$$

- b) If f' is Lipschitz continuous in a neighbourhood of x^* then there exists $K > 0$ such that

$$\|x^{(k+1)} - x^*\| \leq K \|x^{(k)} - x^*\|^2.$$

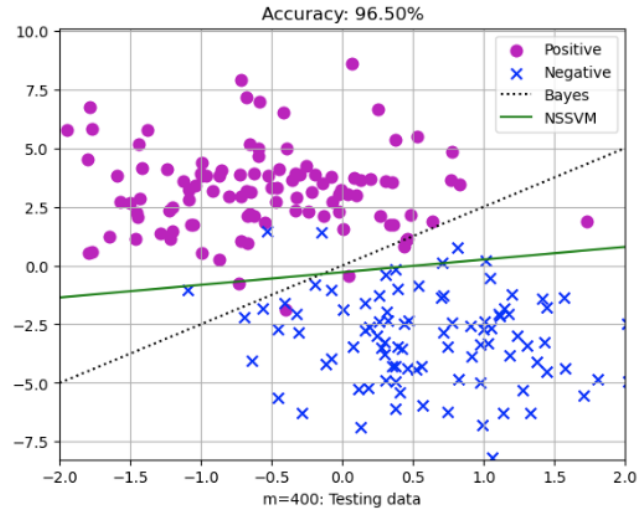
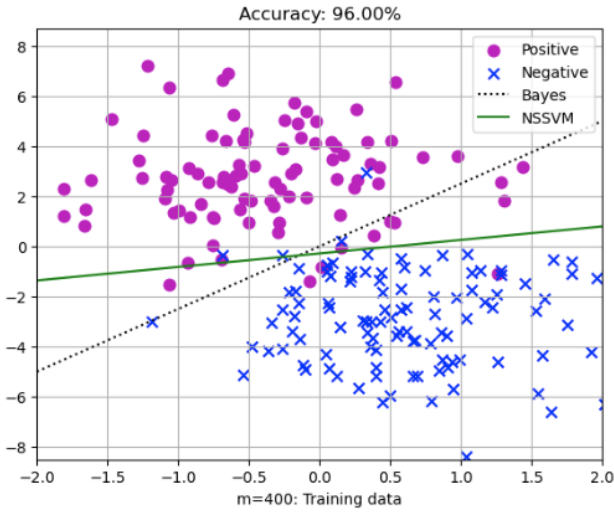


Figure 1: Performance of the NSSVM Algorithm on Binary Classification. The left panel shows the synthetic training data ($m=400$) with an accuracy of 96.00%, illustrating the decision boundary and margin. The right panel demonstrates the model's generalization on testing data ($m=400$) with a comparable accuracy of 96.50%. Both panels highlight the algorithm's efficacy in distinguishing between positive (circles) and negative (crosses) classes.

A) of the theorem indicates that this method has super-linear local convergence. Moreover, from the gradient and Hessian of this sparsity constrained kernel optimization problem, indicates that the derivative is Lipschitz continuous as given in Zhou [2]'s proof of Theorem 2.1. This is better than linear convergence: $\|x^{(k)} - x^*\|$ linearly $\rightarrow 0$ implies $\frac{\|x^{(k+1)} - x^*\|}{\|x^{(k)} - x^*\|} \leq c\|x^{(k)} - x^*\|$ for some $c \in (0, 1)$ [18].

3 Solution and Discussion to Problem

3.1 Solution to Sparsity Constrained Kernel Optimization Problem

The relevant parameter choices involves 4 parameters which are as listed in Table 2 below. There are 2 cases to consider as we are presenting two types of datasets, which are synthetic data ($0 \leq m \leq 10^4$) and real data which have $m > 10^4$. Here, from Fig-

Table 2: Relevant Parameters Choices for NSSVM

cases	C	c	s_0	η
$m \leq 10^4$	0.25	0.0025	s(0.5)	$\frac{1}{m}$
$m > 10^4$	0.25	0.0025	s(1)	$\frac{1}{m}$

ure 1, we applied 2 solves (NSSVM and Bayes classifier) for solving. Overall, all solvers can classify the dataset well. While we cannot derive all combinations of these 4 parameters, we use the parameters above, such as standardised in the paper. The obtained solutions α^* of problem shown in Appendix 4.4.1 has non-zero values which suggest that these are support vectors that define the margin. The sign of α^* (positive or negative) is associated with class

of support vector. The presence of significant number of 0 coefficients suggest that many of data points are not support vectors and do not directly affect decision boundary. The model seems to have achieved high level of accuracy based of Figure 1. Relatively close accuracy between training and testing suggest there is no significant overfitting. The number of zero coefficients could be indicative of potential misclassification that a more complex model or kernel might capture.

3.2 Convergence Plots

From Figure 2, the plot for the synthetic dataset shows some fluctuations which indicates instability in convergence. Trend does not consistently appear to be linear, superlinear or quadratic. This could be due to different factors such as numerical instability, or ill-conditioning. The plot for the real dataset shows a more smooth convergence. Towards the end, the curve steepens which could indicate quadratic convergence as the error decreases faster as it approaches the solution. Ideally, we need to look for a more pronounced bending to determine if convergence is quadratic.

3.3 Relevant Checks

Based on Figure 2, it did not enjoy one-step convergence argued theoretically in Section 2.5 as seen in the increasing number of iterations required for error to decrease. However, that could just be because the implementation did not have a starting point in the local area of η -starting point. Theoretically, this is possibly because the starting point and certain parameters did not meet the conditions justified for hav-

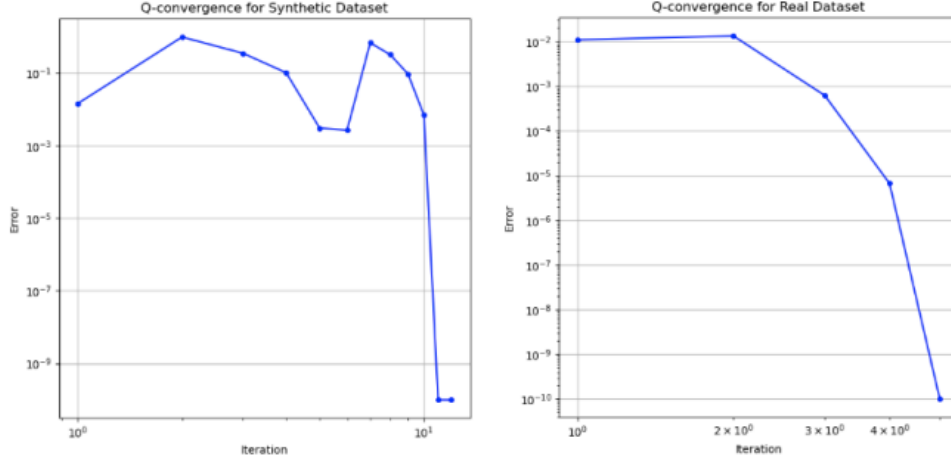


Figure 2: Convergence plots

ing one -step convergece. However, we expected to have local quadratic convergence, typical of different Newton methods which was indicative by our Q-convergence plots as mentioned above. Global convergence was not attained, as we would need a more stable and consistent decrease towards 0. However, we needed to have more checks regarding global convergence. Step size 1 was not attained and trust region was not active as checked. Therefore, global convergence was not reached.

3.4 Performance of Subspace Newton Method

From Zhou [2]’s paper, which discusses the Newton direction derivation here:

$$\begin{aligned} d_{m+1}^k &= -\frac{\langle \mathbf{y}_{T_k}, \Theta^{-1} g_{T_k}(\mathbf{z}^k) - \mathbf{P}_{T_k}^k \rangle}{\langle \mathbf{y}_{T_k}, \Theta^{-1} \mathbf{y}_{T_k}^k \rangle} \\ \mathbf{d}_{T_k}^k &= -\Theta^{-1} \left[g_{T_k}(\mathbf{z}^k) + d_{m+1}^k \mathbf{y}_{T_k} \right] \\ \mathbf{d}_{T_k}^k &= -\mathbf{P}_{T_k}^k \end{aligned} \quad (9)$$

where $\Theta := H_{T_k}(\mathbf{P}^k)$. We can see that calculations of Θ^{-1} and \mathbb{T}_s take up most of the computational complexity. Zhou [2] concluded that overall, NSSVM’s total complexity for each step is:

$$O(mn + \min\{n, s\}s^2)$$

This total complexity can be derived from this analysis here;

1. Matrix Operations, the calculation of Θ , based on Hessian Matrix H_{T_k} of optimization problem, contributes to overall computational complexity. Cost of computation is given by $O_{max}\{n, s\}, s^2\}$.

2. Matrix Inversion, Once Θ is computed, inverting it is necessary to process the algorithm, which carries a complexity of $O(s^3)$ / Cubic cost can become significant for larger s .
3. Subset Selection: Another step in selecting a subset T_{k+1} from a larger set \mathbb{T}_s , based on computation of function $g(z^{k+1})$ and selection of k largest elements influenced by the computation. Complexity is $O(mn)$ as we need to compute the product of Q and vector, and selecting largest elements, which is costly.

Compared to other methods in Table 3, it is known that second order methods such as TensorSVMs[19] and Low Rank Kernel SVMs [20] can converge quickly within small number of iterations. The first few methods in Table are first-order methods. Based

Table 3: Complexity of different algorithms: First 5 are first-order methods and last 4 are second-order methods. Zhou [2] has proven that NSSVM has the lowest computational complexity among 2nd Order methods

Method	Complexity
Reduced SVM [21]	$O(mn + N_k m^2)$
Sparse SVM* [7]	$O(mn)$
Condensed SVM [22]	$O(m_k^3 + mm_k)$
Sparse SVM* [23]	$O(N_k m_n + n \ln(B))$
Sparse SVM* [24]	$O(N_k m^2)$
Low Rank Kernel SVMs	$O(mr^2)$
TensorSVM [19]	$O(m^2(n + r))$
Sparse SVM* [25]	$O(m^3)$
NSSVM	$O(mn + \max\{n, s^2\})$

on CPU time, memory usage and number of SVMs used (Table 4), NSSVM algorithm shows promising scalability in CPU time across different dataset sizes,

with only a slight increase as the size grows. Memory usage, however, escalates notably for larger datasets, implying a higher demand for computational resources. The number of SVMs used also increases, potentially reflecting greater model complexity for larger datasets. Training was implemented on a Intel UHD Graphics 617 1536 MB. From Table 4,

Table 4: Resource Utilization of the NSSVM Algorithm Across Different Dataset Sizes, showing the computational requirements in terms of CPU time and memory alongside the count of support vectors involved.

Dataset Size	CPU time (s)	Memory Used (MiB)	Number of SVMs used
m=1000 (500×2)	0.040	2.543	61
m=2000 (1000×2)	0.048	7.695	78
m=4000 (2000×2)	0.041	30.47	96
m=8000 (4000×2)	0.337	122.4	116
RealDataset (53602×17)	0.415	2070	1104

NSSVM’s performance exhibits scaling challenges as the dataset size increases, with the number of SVMs used and memory requirements rising substantially, while CPU time grows moderately. Memory usage jumps significantly for the larger RealDataset, indicating a possible nonlinear relationship between dataset size and memory consumption. Despite the increased data points and features in the RealDataset, the CPU time does not rise proportionately, suggesting some efficiency in handling feature space. Overall, the method seems to have higher computational demands for larger datasets, both in terms of memory and processing power.

3.5 Classification Result

The classification results displayed in the confusion matrix show high accuracy for both training and testing datasets, with the model achieving over 95% accuracy in most cases. For the training data with 400 samples, the model correctly identified 129 samples as positive and 96 as negative, with few misclassifications. Testing data with the same number of samples also showed similar high accuracy, indicating good model generalization. However, for the larger RealDataset, there was a notable drop in performance, suggesting potential overfitting or a need for model re-tuning.

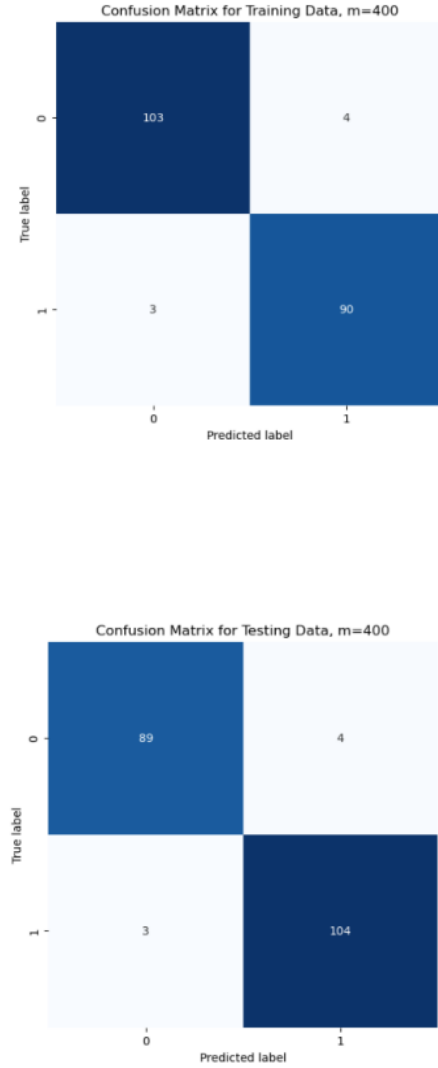


Figure 3: Classification Result: NSSVM Confusion Matrix for m=400 sample size for training dataset and test dataset

Table 5: Accuracy Comparison of the NSSVM Algorithm for Varying Sizes of Datasets, presenting the training and testing accuracy percentages, highlighting the model’s performance from smaller to larger and more complex datasets

Dataset Size	Training Accuracy (%)	Test Accuracy (%)
m=1000 (500×2)	95.80	97.40
m=2000 (1000×2)	97.20	96.70
m=4000 (2000×2)	96.80	96.50
m=8000 (4000×2)	96.55	96.67
RealDataset (53602×17)	82.50	82.74

References

- [1] Cortes, Corinna and Vapnik, Vladimir. ‘Support-vector networks’. In: *Machine Learning* 20.3 (1995), 273–297. DOI: <https://doi.org/10.1007/bf00994018>. URL: <https://link.springer.com/article/10.1007/bf00994018>.
- [2] Zhou, Shenglong. ‘Sparse SVM for Sufficient Data Reduction’. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), 1–1. DOI: <https://doi.org/10.1109/tpami.2021.3075339>. URL: <https://arxiv.org/pdf/2005.13771.pdf>.
- [3] Huang, Xiaolin, Shi, Lei and Suykens, Johan A. K. ‘Support Vector Machine Classifier With Pinball Loss’. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36 (May 2014), pp. 984–997. DOI: [10.1109/tpami.2013.178](https://doi.org/10.1109/tpami.2013.178). (Visited on 15/05/2022).
- [4] Jumutc, Vilen, Huang, Xiaolin and Johan. ‘Fixed-size Pegasos for hinge and pinball loss SVM’. In: *Lirias (KU Leuven)* (Aug. 2013). DOI: [10.1109/ijcnn.2013.6706864](https://doi.org/10.1109/ijcnn.2013.6706864). (Visited on 21/04/2024).
- [5] Gestel, Tony van et al. ‘Benchmarking Least Squares Support Vector Machine Classifiers’. In: *Machine Learning* 54 (Jan. 2004), pp. 5–32. DOI: [10.1023/b:mach.0000008082.80494.e0](https://doi.org/10.1023/b:mach.0000008082.80494.e0). (Visited on 09/09/2019).
- [6] Freund, Yoav and Schapire, Robert E. ‘A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting’. In: *Journal of Computer and System Sciences* 55 (1997), pp. 119–139. DOI: [10.1006/jcss.1997.1504](https://doi.org/10.1006/jcss.1997.1504). URL: <http://www.face-rec.org/algorithms/Boosting-Ensemble/decision-theoretic-generalization.pdf> (visited on 05/12/2019).
- [7] Cotter, Andrew, Shalev-Shwartz, Shai and Srebro, Nati. ‘Learning Optimally Sparse Support Vector Machines’. In: (June 2013), pp. 266–274. (Visited on 21/04/2024).
- [8] Bi, Jinbo, Chen, Yixin and Wang, James. *A Sparse Support Vector Machine Approach to Region-Based Image Categorization*. URL: https://jinbo-bi.uconn.edu/wp-content/uploads/sites/2638/2018/12/cvpr05_mil.pdf (visited on 21/04/2024).
- [9] Awad, Mariette and Khanna, Rahul. ‘Support Vector Machines for Classification’. In: *Efficient Learning Machines* (2015), pp. 39–66. DOI: [10.1007/978-1-4302-5990-9_3](https://doi.org/10.1007/978-1-4302-5990-9_3).
- [10] Zhang, Weizhong et al. ‘Scaling Up Sparse Support Vector Machines by Simultaneous Feature and Sample Reduction’. In: (July 2016), pp. 4016–4025. (Visited on 21/04/2024).
- [11] Fuji, Terunari, Poirion, Pierre-Louis and Takeda, Akiko. *RANDOMIZED SUBSPACE REGULARIZED NEWTON METHOD FOR THE UNCONSTRAINED NON-CONVEX OPTIMIZATION* *. URL: <https://arxiv.org/pdf/2209.04170.pdf> (visited on 21/04/2024).
- [12] Zhou, Shenglong, Pan, Lili and Xiu, Naihua. ‘Newton method for 0-regularized optimization’. In: *Numerical Algorithms* (Mar. 2021). DOI: [10.1007/s11075-021-01085-x](https://doi.org/10.1007/s11075-021-01085-x). (Visited on 08/11/2021).
- [13] Gower, Robert M. et al. *RSN: Randomized Subspace Newton*. arXiv.org, Oct. 2019. DOI: [10.48550/arXiv.1905.10874](https://doi.org/10.48550/arXiv.1905.10874). URL: <https://arxiv.org/abs/1905.10874> (visited on 21/04/2024).
- [14] Duan, Hua et al. ‘An incremental learning algorithm for Lagrangian support vector machines’. In: *Pattern Recognition Letters* 30 (Nov. 2009), pp. 1384–1391. DOI: [10.1016/j.patrec.2009.07.006](https://doi.org/10.1016/j.patrec.2009.07.006). (Visited on 05/03/2021).
- [15] Jiang, Xiufeng, Yi, Zhang and Lv, Jian Cheng. ‘Fuzzy SVM with a new fuzzy membership function’. In: *Neural Computing and Applications* 15 (Feb. 2006), pp. 268–276. DOI: [10.1007/s00521-006-0028-z](https://doi.org/10.1007/s00521-006-0028-z). (Visited on 26/11/2019).
- [16] Siegel, Carl Ludwig. ‘A Mean Value Theorem in Geometry of Numbers’. In: *Annals of mathematics* 46 (Apr. 1945), pp. 340–340. DOI: [10.2307/1969027](https://doi.org/10.2307/1969027). (Visited on 21/04/2024).
- [17] Pan, LiLi, Xiu, NaiHua and Fan, Jun. ‘Optimality conditions for sparse nonlinear programming’. In: *Science China Mathematics* 60 (Mar. 2017), pp. 759–776. DOI: [10.1007/s11425-016-9010-x](https://doi.org/10.1007/s11425-016-9010-x). (Visited on 08/11/2021).
- [18] Pena, Javier. *Convex Optimization*. Cmu.edu, 2016. URL: <https://www.stat.cmu.edu/~ryantibs/convexopt-F16/> (visited on 21/04/2024).

- [19] Zhang, Shaoshuai, Shah, Ruchi and Wu, Panruo. 'TensorSVM'. In: (June 2020). DOI: [10 . 1145 / 3392717 . 3392770](https://doi.org/10.1145/3392717.3392770). (Visited on 21/04/2024).
- [20] Fine, Shai et al. 'Efficient SVM Training Using Low-Rank Kernel Representations'. In: *Journal of Machine Learning Research* 2 (2001), pp. 243–264. URL: [https : / / www . jmlr . org / papers / volume2 / fine01a / fine01a . pdf](https://www.jmlr.org/papers/volume2/fine01a/fine01a.pdf) (visited on 21/04/2024).
- [21] Lee, Yuh-Jye and Mangasarian, Olvi L. 'RSVM: Reduced Support Vector Machines'. In: *Proceedings of the 2001 SIAM International Conference on Data Mining* (Apr. 2001). DOI: [10 . 1137 / 1 . 9781611972719 . 13](https://doi.org/10.1137/1.9781611972719.13). (Visited on 20/06/2022).
- [22] Nguyen, Dung Duc et al. 'Condensed Vector Machines: Learning Fast Machine for Large Data'. In: *IEEE Transactions on Neural Networks* 21 (Dec. 2010), pp. 1903–1914. DOI: [10 . 1109 / tnn . 2010 . 2079947](https://doi.org/10.1109/tnn.2010.2079947). (Visited on 31/08/2020).
- [23] Tan, Mingkui, Wang, Li and Tsang, Ivor W. 'Learning Sparse SVM for Feature Selection on Very High Dimensional Datasets'. In: (June 2010), pp. 1047–1054. (Visited on 21/04/2024).
- [24] Liu, Zhenqiu, Elashoff, David and Piantadosi, Steven T. 'Sparse support vector machines with L0 approximation for ultra-high dimensional omics data'. In: *Artificial intelligence in medicine* 96 (May 2019), pp. 134–141. DOI: [10 . 1016 / j . artmed . 2019 . 04 . 004](https://doi.org/10.1016/j.artmed.2019.04.004). (Visited on 21/04/2024).
- [25] Lázaro, Jorge López et al. 'Sparse LS-SVMs with L0-norm minimization'. In: (Jan. 2011), pp. 189–194. (Visited on 21/04/2024).

4 Appendix

4.1 Proof of Theorems

4.1.1 Global minimizers of Sparsity Constrained Kernel Optimization Problem

We can conclude that it admits a global solution/minimizer. The global minimizers of Equation 3 exist. Proof The solution set is non-empty since 0 satisfies the constraints of 3. The problem can be written as:

$$\min_{|T| \leq s, T \subseteq \mathbb{N}_m} \left\{ \min_{\alpha \in \mathbb{R}^m} D(\alpha) : \langle \alpha_T, \mathbf{y}_T \rangle = 0 \right\}.$$

$D(\cdot)$ is strongly convex. Therefore, the inner problem is a strongly convex program which admits a unique solution, say α_T . In addition, the choices of T such that $|T| \leq s, T \subseteq \mathbb{N}_m$ are finitely many. To derive the global optimal solution, we just pick one T from those choices making $D(\alpha_T)$ the smallest.

4.1.2 Proof of One-step Convergence (Zhou (2021))

Consider a point $\mathbf{z}_t^k = \mathbf{z}^* + t(\mathbf{z}^k - \mathbf{z}^*)$ with $t \in [0, 1]$. Since $\mathbf{z}^k \in U(\mathbf{z}^*, \delta^*)$, it also holds $\mathbf{z}_t^k \in U(\mathbf{z}^*, \delta^*)$ because of

$$\|\mathbf{z}_t^k - \mathbf{z}^*\| = t \|\mathbf{z}^k - \mathbf{z}^*\| \leq \|\mathbf{z}^k - \mathbf{z}^*\| < \delta^*.$$

We first prove that

$$E(\alpha^k) = E(\alpha_t^k).$$

In fact, if $\alpha^* = 0$, then $\alpha_t^k = t\alpha^k$, which means α_t^k and α^k have the same signs. This together with the definition (1.13) of $E(\cdot)$ shows (3.24) immediately. If $\alpha^* \neq 0$, then same reasoning proving (3.14) also derives that

$$\begin{aligned} \alpha_i^* > 0 &\implies \alpha_i^k > 0, & (\alpha_t^k)_i &= (1-t)\alpha_i^* + t\alpha_i^k > 0, \\ \alpha_i^* < 0 &\implies \alpha_i^k < 0, & (\alpha_t^k)_i &= (1-t)\alpha_i^* + t\alpha_i^k < 0, \\ \alpha_i^* = 0 &\implies & (\alpha_t^k)_i &= t\alpha_i^k. \end{aligned}$$

These also mean α_t^k and α^k have the same signs. So (3.24) is true and brings out

$$H(\alpha^k) \stackrel{(1.12)}{=} E(\alpha^k) + Q^\top Q \stackrel{(3.24)}{=} E(\alpha_t^k) + Q^\top Q = H(\alpha_t^k).$$

Then for any $T_k \in \mathbb{T}_s(\alpha^k - \eta g(\mathbf{z}^k))$, the above equation contributes to

$$\begin{aligned} \nabla F(\mathbf{z}_t^k; T_k) &= \begin{bmatrix} H_{T_k}(\alpha_t^k) & 0 & \mathbf{y}_{T_k} \\ 0 & I & 0 \\ \mathbf{y}_{T_k}^\top & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} H_{T_k}(\alpha^k) & 0 & \mathbf{y}_{T_k} \\ 0 & I & 0 \\ \mathbf{y}_{T_k}^\top & 0 & 0 \end{bmatrix} = \nabla F(\mathbf{z}^k; T_k). \end{aligned}$$

It follows from the mean value theorem that there exists a \mathbf{z}_t^k satisfying

$$\begin{aligned} F(\mathbf{z}^k; T_k) &\stackrel{(3.11)}{=} F(\mathbf{z}^k; T_k) - F(\mathbf{z}^*; T_k) \\ &= \nabla F(\mathbf{z}_t^k; T_k) (\mathbf{z}^k - \mathbf{z}^*) \\ &= \nabla F(\mathbf{z}^k; T_k) (\mathbf{z}^k - \mathbf{z}^*) \end{aligned}$$

which together with $\nabla F(\mathbf{z}^k; T_k)$ being always non-singular because of (2.18) suffices to

$$\begin{aligned} \mathbf{z}^* &= \mathbf{z}^k - \left(\nabla F(\mathbf{z}^k; T_k) \right)^{-1} \nabla F(\mathbf{z}^k; T_k) (\mathbf{z}^k - \mathbf{z}^*) \\ &\stackrel{(3.1)}{=} \mathbf{z}^k + \mathbf{d}^k \stackrel{(3.3)}{=} \mathbf{z}^{k+1}. \end{aligned}$$

Finally, for any $T_{k+1} \in \mathbb{T}_s(\alpha^{k+1} - \eta g(\mathbf{z}^{k+1})) = \mathbb{T}_s(\alpha^* - \eta g(\mathbf{z}^*))$, it follows from \mathbf{z}^* being an η -stationary point that

$$\|F(\mathbf{z}^{k+1}, T_{k+1})\| = \|F(\mathbf{z}^*, T_{k+1})\| \stackrel{(2.17)}{=} 0.$$

4.1.3 Convergence Analysis

Before the main convergence property, we define some constants

$$\begin{aligned}\gamma &:= 2 \max \{1 + \eta/c + \eta\|Q\|^2, \eta\sqrt{m}\}, \\ \eta^* &:= \begin{cases} \|\alpha^*\|_{[s]} \|\mathbf{g}(\mathbf{z}^*)\|_\infty^{-1}, & \text{if } \|\alpha^*\|_0 = s, \\ +\infty, & \text{if } \|\alpha^*\|_0 < s, \end{cases} \\ \delta^* &:= \begin{cases} \gamma^{-1} (\|\alpha^*\|_{[s]} - \eta\|\mathbf{g}(\mathbf{z}^*)\|_\infty), & \text{if } \|\alpha^*\|_0 = s, \\ \gamma^{-1} \min_{i \in S_*} |\alpha_i^*|, & \text{if } 0 < \|\alpha^*\|_0 < s, \\ +\infty, & \text{if } \|\alpha^*\|_0 = 0. \end{cases}\end{aligned}$$

Based on which, we first present some properties regarding an η -stationary point of (1.4). Lemma 3.1 Let \mathbf{z}^* be an η -stationary point of (1.4) for some $0 < \eta < \eta^*$, and η^* and δ^* be given by (3.8) and (3.9). Then for any $\mathbf{z} \in \mathbf{U}(\mathbf{z}^*, \delta^*)$, we have the following results. a) The parameters $\eta^* > 0$ and $\delta^* > 0$. b) For any $\mathbf{T} \in \mathbb{T}_s(\alpha - \eta\mathbf{g}(\mathbf{z}))$ and any $\mathbf{T}_* \in \mathbb{T}_s(\alpha^* - \eta\mathbf{g}(\mathbf{z}^*))$, it holds

$$\begin{cases} S_* = \mathbf{T}_* = \mathbf{T} = \text{supp}(\alpha), & \text{if } \|\alpha^*\|_0 = s, \\ S_* \subseteq (\mathbf{T}_* \cap \mathbf{T} \cap \text{supp}(\alpha)), & \text{if } \|\alpha^*\|_0 < s. \end{cases}$$

c) For any $\mathbf{T} \in \mathbb{T}_s(\alpha - \eta\mathbf{g}(\mathbf{z}))$, it holds

$$\mathbf{F}(\mathbf{z}^*; \mathbf{T}) = 0.$$

4.2 Datasets

We consider two datasets, one synthetic data and real data in higher dimensions. When comparing the performance of the method, let α be the classifier generated by the method. We consider two-dimensional examples with the generated synthetic data. The `randomData` function generates these datasets. In the 2D case:

- Generates a vertically stacked array of two Gaussian distributions with $\frac{m}{2}$ samples each.
- For positive class ($c=1$), the datapoints are drawn from a Gaussian distribution centered at (0.5,-3) with variance (0.5,3).
- For negative class ($c=-1$), the data points are drawn from a Gaussian distribution centered at (-0.5,3) with variance (0.5,3).
- The result is 2 clusters with class labels $\{1, -1\}$, located diagonally across from each other on the Cartesian plane.

The 3D case as follows:

- Generates two classes of 3D data with $\frac{m}{2}$ samples each.
- The positive class is generated in cylindrical coordinates (ρ, θ, z) with ρ centered around 0.5 and perturbed by a small Gaussian noise with standard deviation 0.03, and θ uniformly distributed between 0 and 2π . The z coordinate is the square of ρ .
- The negative class is generated similarly but with z as the negative square of ρ . This creates a 3D distribution where one class is above the plane and the other is mirrored below, both centered around a cylinder axis.

It is good to note that the real dataset is described and detailed by Zhou [2]. It has 17 dimensions, and consists of a manually curated coagulation of different datasets that are inseparable.

4.3 Code

Alternatively, full repository, dataset and code can be found here. <https://github.com/anabelyong/NSSVM-python>.

4.3.1 NSSVM Implementation

```
import numpy as np
from copy import deepcopy
from scipy.sparse import issparse, diags
import time

class NSSVM:
    def __init__(self):
        pass

    def fit(self, X, y, pars=None):
        self.X = X
        self.y = y
        self.m, self.n = X.shape
        out = self._NSSVM(X, y, pars)

        # for predict
        self.w = out['w'][:-1]
        self.b = out['w'][-1]

        return out

    def predict(self, X):
        return np.sign(X @ self.w + self.b)

    def _Fnorm(self, var):
        return np.linalg.norm(var, 2) ** 2

    #Newton method for sparse SVMs with sparsity level tuning
    def _gradient_descent(self, fun, x0, lr=0.01, tol=1e-6, max_iter=100):
        x = x0
        for _ in range(max_iter):
            grad = self._approximate_gradient(fun, x)
            x_new = x - lr * grad
            if np.linalg.norm(x_new - x) < tol:
                break
            x = x_new
        return x

    def _approximate_gradient(self, fun, x, eps=1e-8):
        grad = np.zeros_like(x)
        for i in range(len(x)):
            x_plus = np.array(x, copy=True)
            x_minus = np.array(x, copy=True)
            x_plus[i] += eps
            x_minus[i] -= eps
            grad[i] = (fun(x_plus) - fun(x_minus)) / (2 * eps)
        return grad

    def _NSSVM(self, X, y, pars):
        t0 = time.time()
        if issparse(X) and np.count_nonzero(X) / X.size > 0.1:
            X = X.toarray()

        if self.n < 3e4:
            Qt = np.diag(y) @ X
        else:
            Qt = diags(y) @ X

        Q = Qt.T
        Fnorm = self._Fnorm

        maxit, alpha, tune, disp, tol, eta, s0, C, c = self._get_parameters(self.m, self.n)
        if pars is not None:
            if 'maxit' in pars: maxit = pars['maxit']
            if 'alpha' in pars: alpha = pars['alpha']
            if 'disp' in pars: disp = pars['disp']
            if 'tune' in pars: tune = pars['tune']
            if 'tol' in pars: tol = pars['tol']
            if 'eta' in pars: eta = pars['eta']
            if 's0' in pars: s0 = min(self.m, pars['s0'])
            if 'C' in pars: C = pars['C']
            if 'c' in pars: c = pars['c']

        T1 = np.where(y == 1)[0]
        T2 = np.where(y == -1)[0]
        nT1 = T1.size
        nT2 = T2.size
```



```

if nT1 < s0:
    T = np.concatenate((T1, T2[:s0 - nT1]))
elif nT2 < s0:
    T = np.concatenate((T1[:s0 - nT2], T2))
else:
    T = np.concatenate((T1[:int(np.ceil(s0 / 2))], T2[:int(s0 - np.ceil(s0 / 2))]))

T = np.sort(T[:s0])
s = s0
b = (nT1 >= nT2) - (nT1 < nT2)
bb = b
w = np.zeros(self.n)
gz = -np.ones(self.m)
ERR = np.zeros(maxit)
ACC = np.zeros(maxit)
ACC[0] = 1 - np.count_nonzero(np.sign(b) - y) / self.m
ET = np.ones(s) / C

maxACC = 0
flag = 1
j = 1
r = 1.1
count = 1
count0 = 2
iter0 = -1

for iter in range(1, maxit + 1):
    if iter == 1 or flag:
        QT = Q[:, T]
        QtT = Qt[T, :]
        yT = y[T]
        ytT = yT.T

        alphaT = alpha[T]
        gzT = -gz[T]
        alyT = -ytT @ alphaT

        err = (np.abs(Fnorm(alpha) - Fnorm(alphaT)) + Fnorm(gzT) + alyT ** 2) / (self.m * self.n)
        ERR[iter - 1] = np.sqrt(err)

        if tune and iter < 30 and self.m <= 1e8:
            stop1 = iter > 5 and err < tol * s * np.log2(self.m) / 100
            stop2 = s != s0 and np.abs(ACC[iter - 1] - np.max(ACC[:iter - 1])) <= 1e-4
            stop3 = s != s0 and iter > 10 and np.max(ACC[iter - 5:iter]) < maxACC
            stop4 = count != count0 + 1 and ACC[iter - 1] >= ACC[0]
            stop = stop1 and (stop2 or stop3) and stop4
        else:
            stop1 = err < tol * np.sqrt(s) * np.log10(self.m)
            stop2 = iter > 4 and np.std(ACC[iter - 2:iter]) < 1e-4
            stop3 = iter > 20 and np.abs(np.max(ACC[iter - 9:iter]) - maxACC) <= 1e-4
            stop = (stop1 and stop2) or stop3

        if disp:
            print(f' {iter:3d} {err:.2e} {ACC[iter - 1]:.5f}')

        if ACC[iter - 1] > 0 and (ACC[iter - 1] >= 0.99999 or stop):
            break

        ET0 = deepcopy(ET)
        ET = (alphaT >= 0) / C + (alphaT < 0) / c

    if min(self.n, s) > 1e3:
        d = self._my_cg(QT, yT, ET, np.concatenate((gzT, [alyT])), 1e-10, 50, np.zeros(s + 1))
        dT = d[:s]
        dend = d[-1]
    else:
        if s <= self.n:
            if iter == 1 or flag:
                PTT0 = QtT @ QT
                PTT = PTT0 + np.diag(ET)
                d = np.linalg.solve(np.concatenate((np.concatenate((PTT, ytT.reshape(-1, 1)), axis=1),
                                                            np.concatenate((ytT.reshape(-1, 1), np.array([[0]])),
                                                            axis=1)), axis=0), np.concatenate(
                    (gzT, [alyT])), use_umfpack=True)
                dT = d[:s]
                dend = d[-1]
            else:
                PTT = PTT0 + np.diag(ET)
                d = np.linalg.solve(np.concatenate((PTT, ytT.reshape(-1, 1)), axis=1),
                    np.concatenate((ytT.reshape(-1, 1), np.array([[0]])), axis=0), np.concatenate(
                    (gzT, [alyT])), use_umfpack=True)
                dT = d[:s]
                dend = d[-1]
        else:
            dT = d[:s]
            dend = d[-1]

```

```

ETinv = 1 / ET
flag1 = np.count_nonzero(ET0) != np.count_nonzero(ET)
flag2 = np.count_nonzero(ET0) == np.count_nonzero(ET) and np.count_nonzero(ET0 - ET) == 0
if iter == 1 or flag or flag1 or not flag2:
    EQtT = diags(ETinv) @ QtT
    P0 = np.eye(self.n) + QT @ EQtT
    Ey = ETinv * yT
    Hy = Ey - EQtT @ (np.linalg.solve(P0, QT @ Ey))
    dend = (gzT @ Hy - alyT) / (ytT @ Hy)
    tem = ETinv * (gzT - dend * yT)
    dT = tem - EQtT @ (np.linalg.solve(P0, QT @ tem))

alpha = np.zeros(self.m)
alphaT = alphaT + dT
alpha[T] = alphaT
b = b + dend

w = QT @ alphaT
Qtw = Qt @ w
tmp = y * Qtw

gz = Qtw - 1 + b * y
ET1 = (alphaT >= 0) / C + (alphaT < 0) / c
gz[T] = alphaT * ET1 + gz[T]

j = iter + 1
ACC[j - 1] = 1 - np.count_nonzero(np.sign(tmp + b) - y) / self.m

if self.m <= 1e7:
    bb = np.mean(yT - tmp[T])
    ACCb = 1 - np.count_nonzero(np.sign(tmp + bb) - y) / self.m
    if ACC[j - 1] >= ACCb:
        bb = b
    else:
        ACC[j - 1] = ACCb
else:
    bb = b

if self.m < 6e6 and ACC[j - 1] < 0.5:
    b0 = self._gradient_descent(lambda t: np.sum((np.sign(tmp + t) - y) ** 2), [bb])
    acc0 = 1 - np.count_nonzero(np.sign(tmp + b0) - y) / self.m
    if ACC[j - 1] < acc0:
        bb = b0
        ACC[j - 1] = acc0

if ACC[j - 1] >= maxACC:
    maxACC = ACC[j - 1]
    alpha0 = alpha
    tmp0 = tmp
    maxwb = np.concatenate((w, [bb]))

T0 = T
mark = 0
if tune and (err < tol or iter % 10 == 0) and iter > iter0 + 2 and count < 10:
    count0 = count
    count += 1
    s = min(self.m, np.ceil(r * s))
    iter0 = iter
    if count > (int(self.m >= 1e6) or (self.n < 3)) + 1 * (self.m < 1e6 and self.n >= 5):
        alpha = np.zeros(self.m)
        gz = -np.ones(self.m)
        mark = 1
else:
    count0 = count

if s != self.m:
    if self.m < 5e8:
        T = np.argsort(np.abs(alpha - eta * gz))[-s:]
    else:
        T = np.argsort(np.abs(alpha - eta * gz))[:-1][:-s]
    if mark:
        nT = np.count_nonzero(y[T] == 1)
        if nT == s:
            if nT2 <= 0.75 * s:
                T = np.concatenate((T[:s - int(nT2 / 2)], T2[:int(nT2 / 2)]))
            else:
                T = np.concatenate((T[:int(s / 4)], T2[:s - int(s / 4)]))
        elif nT == 0:

```

```

        if nT1 <= 0.75 * s:
            T = np.concatenate((T[:s - int(nT1 / 2)], T1[:int(nT1 / 2)]))
        else:
            T = np.concatenate((T[:int(s / 4)], T1[s - int(s / 4)]))
        T = np.sort(T)

    else:
        T = np.arange(self.m)
        flag = 1
        flag3 = np.count_nonzero(T0) == s

    if flag3:
        flag3 = np.count_nonzero(T - T0) == 0
    if flag3 or np.count_nonzero(T0) == self.m:
        flag = 0
        T = T0

    wb = np.concatenate((w, [bb]))
    acc = ACC[j - 1]

    if self.m <= 1e7 and iter > 1:
        b0 = self._gradient_descent(lambda t: np.linalg.norm(np.sign(tmp0 + t[0]) - y), [maxwb[-1]])
        acc0 = 1 - np.count_nonzero(np.sign(tmp0 + b0) - y) / self.m
        if acc < acc0:
            wb = np.concatenate((maxwb[:-1], b0))
            acc = acc0

    if acc < maxACC - 1e-4:
        alpha = alpha0
        wb = maxwb
        acc = maxACC

    out = {'s': s,
          'w': wb,
          'sv': s,
          'ACC': acc,
          'iter': iter,
          'time': time.time() - t0,
          'alpha': alpha
          }

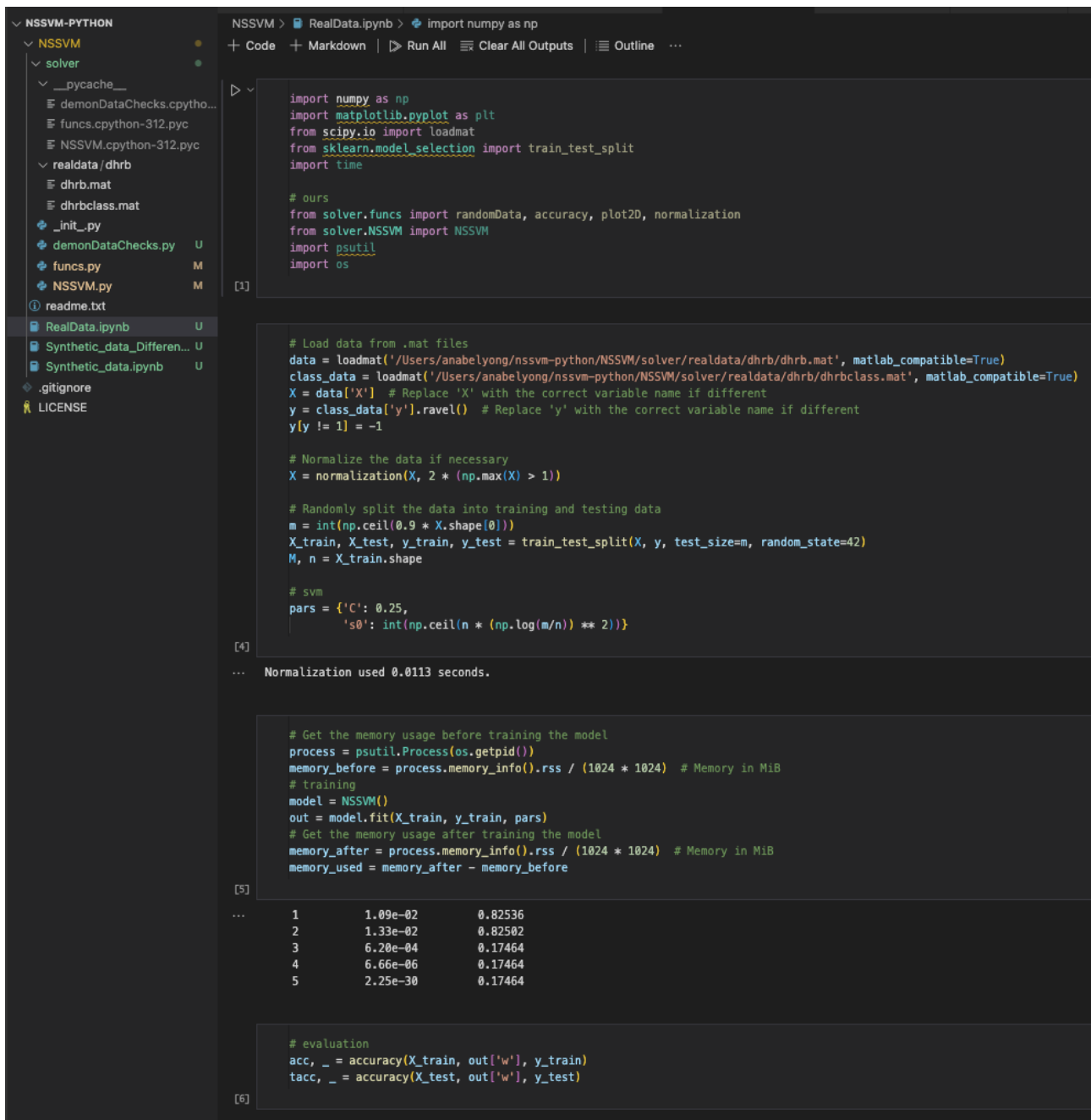
    return out

def _get_parameters(self, m, n):
    maxit = 1000
    alpha = np.zeros(m)
    tune = 0
    disp = 1
    tol = 1e-6
    eta = min(1 / m, 1e-4)
    beta = 1 if max(m, n) < 1e4 else 0.05 if m <= 5e5 else 10
    s0 = int(np.ceil(beta * n * (np.log2(m / n)) ** 2))
    C = np.log10(m) if m > 5e6 else 1 / 2
    c = C / 100
    return maxit, alpha, tune, disp, tol, eta, s0, C, c

def _my_cg(self, Q, y, E, b, cg_tol, cg_iter, x):
    r = b
    e = np.sum(r * r)
    t = e
    for i in range(cg_iter):
        if e < cg_tol * t:
            break
        if i == 1:
            p = r
        else:
            p = r + (e / e0) * p
        p1 = p[:-1]
        w = ((Q @ p1).T @ Q).T + E * p1 + p[-1] * y
        a = e / np.sum(p * w)
        x = x + a * p
        r = r - a * w
        e0 = e
        e = np.sum(r * r)
    return x

```

4.3.2 How to run and visualise NSSVM Binary Classification and Results



The screenshot displays a Jupyter Notebook interface with a file explorer on the left and a code editor on the right. The file explorer shows a directory structure for 'NSSVM-PYTHON' with subdirectories 'NSSVM' and 'solver', and files like 'RealData.ipynb', 'Synthetic_data_Differen...', 'Synthetic_data.ipynb', '.gitignore', and 'LICENSE'.

The code editor shows the following code blocks:

```
[1] import numpy as np
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.model_selection import train_test_split
import time

# ours
from solver.funcs import randomData, accuracy, plot2D, normalization
from solver.NSSVM import NSSVM
import psutil
import os
```

```
[4] # Load data from .mat files
data = loadmat('/Users/anabelyong/nssvm-python/nssvm/solver/realdata/dhrb/dhrb.mat', matlab_compatible=True)
class_data = loadmat('/Users/anabelyong/nssvm-python/nssvm/solver/realdata/dhrb/dhrbclass.mat', matlab_compatible=True)
X = data['X'] # Replace 'X' with the correct variable name if different
y = class_data['y'].ravel() # Replace 'y' with the correct variable name if different
y[y != 1] = -1

# Normalize the data if necessary
X = normalization(X, 2 * (np.max(X) > 1))

# Randomly split the data into training and testing data
m = int(np.ceil(0.9 * X.shape[0]))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=m, random_state=42)
M, n = X_train.shape

# svm
pars = {'C': 0.25,
        's0': int(np.ceil(n * (np.log(m/n)) ** 2))}
```

```
... Normalization used 0.0113 seconds.
```

```
[5] # Get the memory usage before training the model
process = psutil.Process(os.getpid())
memory_before = process.memory_info().rss / (1024 * 1024) # Memory in MiB
# training
model = NSSVM()
out = model.fit(X_train, y_train, pars)
# Get the memory usage after training the model
memory_after = process.memory_info().rss / (1024 * 1024) # Memory in MiB
memory_used = memory_after - memory_before
```

```
...
1      1.09e-02      0.82536
2      1.33e-02      0.82502
3      6.20e-04      0.17464
4      6.66e-06      0.17464
5      2.25e-30      0.17464
```

```
[6] # evaluation
acc, _ = accuracy(X_train, out['w'], y_train)
tacc, _ = accuracy(X_test, out['w'], y_test)
```

Figure 4: Jupyter Notebook Results Logging Part 1

```

• # logging
if hasattr(model, 'alpha'):
    print(f"Support Vector Solutions, alpha^*: {model.alpha}")

# If 'alpha' is part of the 'out' dictionary
if 'alpha' in out:
    print(f"Support Vector Solutions, alpha^*: {out['alpha']}")
print(f"Training Time: {out['time']:.3f}sec")
print(f"Training Size: {m}x{n}")
print(f"Memory Used: {-100*memory_used:.6f} MiB")
print(f"Training Accuracy: {acc*100:.2f}%")
print(f"Testing Size: {X_test.shape[0]}x{n}")
print(f"Testing Accuracy: {tacc*100:.2f}%")
print(f"Number of Support Vectors: {out['sv']}")

```

[6] ✓ 0.2s

```

... Support Vector Solutions, alpha^*: [0.46719251 0.47138682 0.47058213 ... 0.      0.      0.      ]
Training Time: 0.614sec
Training Size: 53602x17
Memory Used: 1759.375000 MiB
Training Accuracy: 82.50%
Testing Size: 53602x17
Testing Accuracy: 82.74%
Number of Support Vectors: 1104

```

```

errors = [
    1.09e-02, 1.33e-02, 6.20e-04, 6.66e-06, 2.25e-30
]

accuracies = [
    0.82536, 0.82502, 0.17464, 0.17464, 0.17464,
]

# Convert to numpy arrays for ease of plotting
iterations = np.arange(1, len(errors) + 1)
errors = np.array(errors)
accuracies = np.array(accuracies)

# Ensure no zero values for log scale
errors = np.maximum(errors, 1e-10)
accuracies = np.maximum(accuracies, 1e-10)

# Create subplots for the semi-log and log-log plots
fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Semi-log plot for error
axs[0].loglog(iterations, errors, 'b-', marker='o', markersize=4)
axs[0].set_xlabel('Iteration')
axs[0].set_ylabel('Error')
axs[0].set_title('Q-convergence for Real Dataset')
axs[0].grid(True)

# Semi-log plot for accuracy
axs[1].semilogy(iterations, 1 - accuracies, 'r-', marker='o', markersize=4) # Plot 1 - accuracy to represent error
axs[1].set_xlabel('Iteration')
axs[1].set_ylabel('1 - Accuracy')
axs[1].set_title('Q-convergence for Real Dataset')
axs[1].grid(True)

plt.tight_layout()
plt.show()

# Log-log plot for error
fig, axs = plt.subplots(1, 2, figsize=(12, 6))

```

Figure 5: Jupyter Notebook Results Logging Part 2

4.4.1 α^* values obtained for Synthetic Dataset with size m=400

Figure 6: α^* Obtained Solution which looks sparse

4.4.2 Supplementary Figures

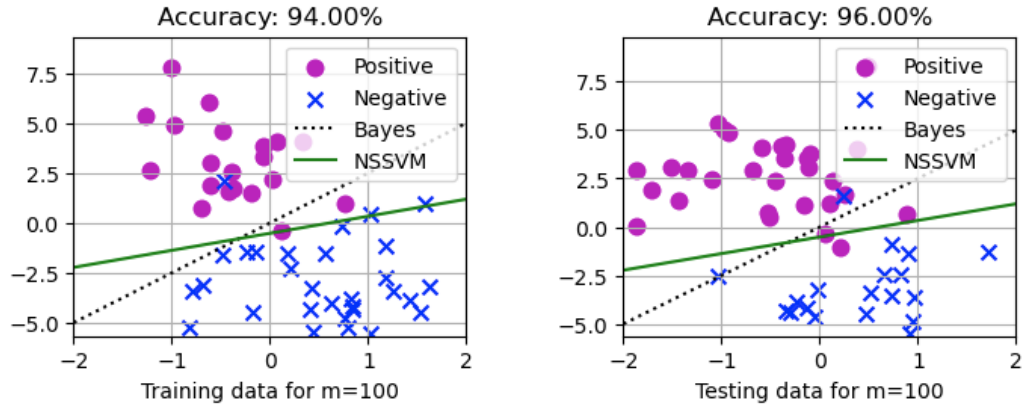


Figure 7: Performance of NSSVM algorithm on Binary Classification with $m=100$ Synthetic Dataset generated

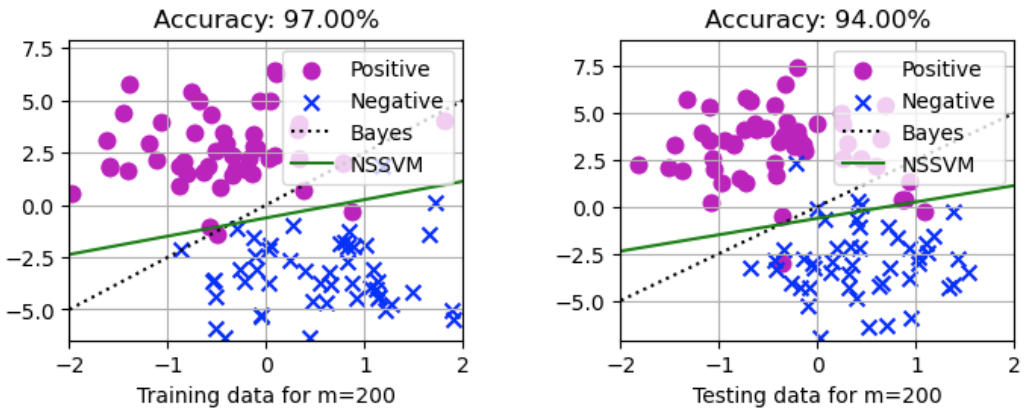


Figure 8: Performance of NSSVM algorithm on Binary Classification with $m=200$ Synthetic Dataset generated

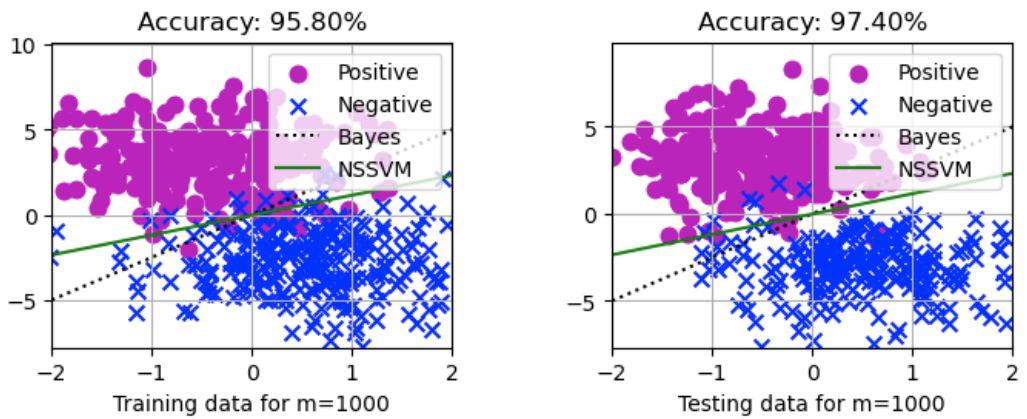


Figure 9: Performance of NSSVM algorithm on Binary Classification with $m=1000$ Synthetic Dataset generated

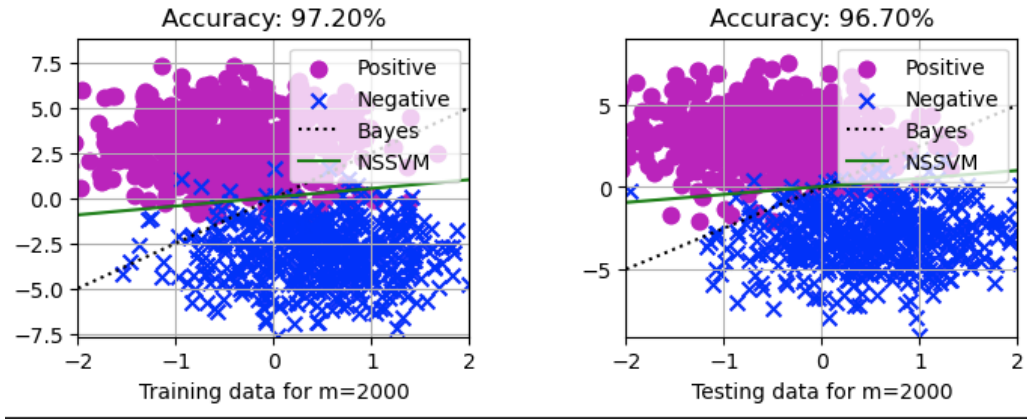


Figure 10: Performance of NSSVM algorithm on Binary Classification with $m=2000$ Synthetic Dataset generated

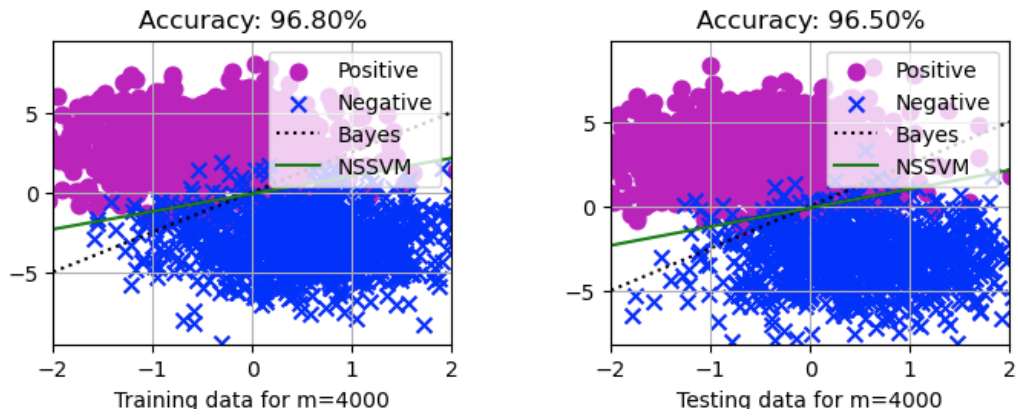


Figure 11: Performance of NSSVM algorithm on Binary Classification with $m=400$ Synthetic Dataset generated

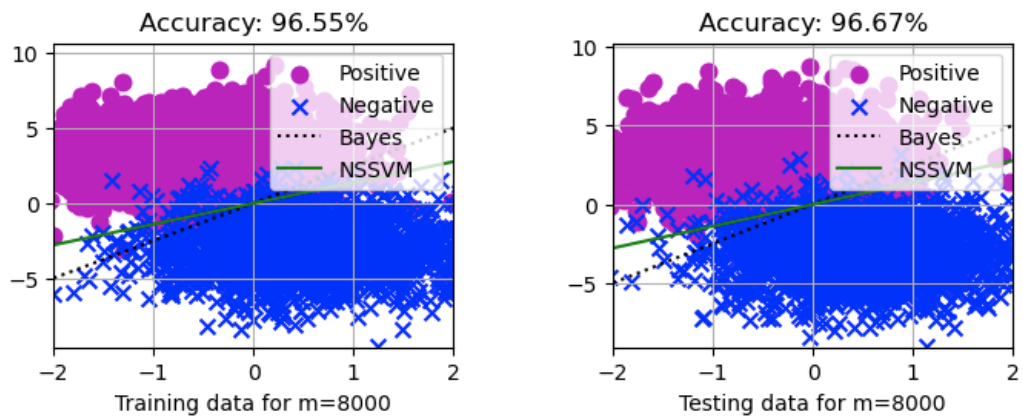


Figure 12: Performance of NSSVM algorithm on Binary Classification with $m=8000$ Synthetic Dataset generated